Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Optimizações numa Linguagem para Desenvolvimento de Aplicações Web

Hélio Dolores (aluno nº 25953)

2º Semestre de 2008/09

29 de Julho de 2009

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Optimizações numa Linguagem para Desenvolvimento de Aplicações Web

Hélio Dolores (aluno nº 25953)

Orientador: Prof. Doutor Luís Caires
Co-orientador: Eng. Lúcio Ferrão

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

2º Semestre de 2008/09

29 de Julho de 2009

Resumo

O desenvolvimento de sistemas para gestão de informação baseada na Web pode ser acelerado através do uso de ferramentas de programação adequadas, permitindo que equipas de programadores rapidamente desenvolvam e disponibilizem soluções eficientes. Aproximações bem sucedidas assentam no uso de linguagens de domínio especifico (DSL), para a construção de algumas camadas dos sistemas. Tipicamente, a lógica aplicacional interage com os dados nos repositórios por meio de queries e comandos SQL. Depois, o código SQL gerado pelo compilador ou interpretador da DSL a partir das descrições fornecidas pelo programador ou utilizador, vai estar geralmente mal estruturado ou sub-optimizado se não houver cuidado da parte do optimizador.

Esta dissertação endereça o desenho e validação de um processo de optimização em tempo de compilação para queries SQL embutidas numa linguagem de domínio especifico, usando a DSL da *OutSystems* como um exemplo motivador. O objectivo não é optimizar a execução de cada query, porque esta tarefa já é bem executada pelo motor da base de dados: a aproximação seguida neste projecto assenta na identificação e análise de padrões sub-optimizados causados pela coexistência entre instruções e comandos SQL e os típicos operadores das linguagens imperativas. De forma a atingir este objectivo utilizamos transformações de código SQL, envolvendo junções e reestruturações de queries; análise de fluxo de dados para lidar com o controlo de dependências entre operações; transformações de programas e técnicas de avaliação parcial, como movimentação de código e pré-cálculo.

Depois, escolhemos um destes padrões para ser completamente implementado no compilador da linguagem *OutSystems*. Esta optimização considera a informação recolhida em tempo de compilação e execução sobre os atributos alterados nas entidades e propaga essa informação para as novas operações de *update* parcial.

Esta dissertação apresenta a implementação e validação, através da análise de aplicações reais, desta optimização, que foi integrada com sucesso e entrará em produção na próxima versão do compilador *OutSystems*.

Palavras-chave: Optimização de compiladores, Análise de fluxo de dados, *OutSystems*, *Update* parcial, Optimização de múltiplas queries, Movimentação de código, Agrupamento de queries

Abstract

The development of web based information management systems may be accelerated through the use of adequate programming tools, enabling teams of programmers to rapidly develop and deploy efficient solutions. Successful approaches rely on the use of domain specific languages (DSL), targeting the several tiers of the system to be constructed. Typically, the application logic interacts with the data repository by means of SQL queries and commands. Then, the SQL code generated by the DSL compiler or interpreter, from the descriptions provided by a programmer or user, will be in general ill-structured and sub-optimal if care is not taken by an optimizer.

This dissertation addresses the design and validation of a compile-time optimization process for SQL queries embedded in a domain specific programming language, using the *OutSystems* DSL as a motivating example. The goal is not to optimize the execution of each query, since this task is already well addressed by the database engine: the approach followed in this project relies on the identification and analysis of sub-optimal patterns caused by the coexistence between SQL instructions and the typical operators of imperative languages. In order to achieve this goal we have used SQL source-to-source transformations, involving merging and restructuring of queries and commands; data-flow analysis to deal with dependency control between operations; program transformation and partial evaluation techniques, like code motion or prefetching.

Afterwards, we have chose one of these patterns to be fully implemented in the *OutSystems* compiler. This optimization takes compile-time and runtime information about entity changed attributes and propagates them to the new partial update operations.

This dissertation presents the implementation and validation, through the analysis of real applications, of this optimization, that was successfully integrated in the next release of the *OutSystems* compiler.

Keywords: Compiler optimizations, Data flow analisys, *OutSystems*, Partial update, Multi-query optimization, Code motion, Query batch

Agradecimentos

Primeiro gostava de agradecer ao Professor Luís Caires e ao Eng. Lúcio Ferrão pelo apoio e orientação neste projecto e pela oportunidade de trabalhar com uma equipa fantástica como é a *OutSystems R&D*. Gostava de lhes agradecer também, assim como ao António Melo, pelo apoio e compreensão quando precisei de algum tempo para dedicar aos meus assuntos pessoais.

Os meus agradecimentos vão também para:

- Toda a equipa da *OutSystems*, principalmente ao Lúcio Ferrão, António Melo, João Rosado, José Caldeira e Rui Eugénio. Aqui tenho que agradecer especialmente ao Leonardo Fernandes pela enorme paciência e acompanhamento intensivo nestes últimos meses. Gostava também de agradecer a disponibilidade para realização de inquéritos às seguintes pessoas: Gonçalo Borrêga, Nuno Teles, Susete Henriques, Tiago Simões, Francisco Menezes e Paulo Garrudo.
- O CITI e DI/FCT pela oportunidade de trabalhar neste projecto com bolsa de investigação.
- O Professor José Alferes pelos esclarecimentos iniciais sobre bases de dados.
- A minha família, especialmente à minha mãe Custódia por todo o esforço nestes últimos anos para que nunca me faltasse nada, ao meu pai José, e à minha irmã Antónia e cunhado Jorge por me terem acolhido nos últimos tempos. Agradeço também ao meu cão pela companhia nos últimos 11 anos.
- Todos os meus amigos e colegas. Pelo menos o nome destes tenho que referir (sem qualquer ordem): Luís Oliveira, Sofia Penim, Danilo Manmohanlal, Bruno Félix, Pedro Chambel, Paulo Carmo, Pablo Vigarinho, André Vinhas, Pedro Ribeiro, Vera Viegas, Ângelo Monteiro, Ricardo Silva, Alexandra Amado, Sofia Gomes, Maria Café, Nuno Luís, Luísa Lourenço, Bernardo Toninho, Rui Rosendo, Mário Pires, Manuel Pimenta, David Navalho, João Soares, Luís Silva, Ana Neca e Clarisse ...
- Os professores, funcionários e colegas que partilharam comigo os corredores e as salas da FCT/UNL ao longo destes anos e que directa, ou indirectamente, contribuíram para a minha formação académica.

A todos,

Muito Obrigado!

Para a minha mãe Custódia, por tudo ...

Conteúdo

Lista de Figuras

Lista de Tabelas

Listagens de Código

1

Introdução

O desenvolvimento de aplicações Web é geralmente um processo complexo e moroso. Muitas das vezes, quando o produto final chega ao mercado, já chega tarde e ainda com bastantes imperfeições. Este tipo de aplicações é também caracterizado por ser muito difícil de estender a novas funcionalidades, tornando-se essa extensão um processo naturalmente lento e de custos elevados.

Estas condições levam a que surjam alternativas ao método de desenvolvimento habitual, que fornecem aos programadores níveis de abstracção mais elevados, com o objectivo de aumentar os níveis de produtividade.

A empresa *OutSystems* [1] pretende responder a este cenário de desenvolvimento de software, apresentando uma ferramenta capaz de responder a este tipo de desafios: a *Agile Platform*. Esta *framework*, oferece condições para simplificar o processo de criação de aplicações Web, juntando numa só aplicação a capacidade de criar, integrar e publicar sistemas para gestão de informação baseada na Web. É possível obter soluções funcionais num curto espaço de tempo e ir acrescentando novos componentes à medida que as necessidades aparecem. Isto permite acelerar a apresentação de versões provisórias e introduzir modificações rápidas nos programas de acordo com o *feedback* dos clientes, aumentando o seu ciclo de vida e flexibilidade.

Por outro lado, o elevado nível do estilo de programação e a existência de graus de especialização muito elevados da parte dos programadores, leva a que o código gerado por tais ferramentas genéricas seja ineficiente. Nesta dissertação, consideramos o problema de optimização do código utilizando a plataforma *OutSystems* como exemplo motivador, onde analisámos cenários sub-optimizados e propomos as devidas melhorias.

1.1 Contexto e Motivação

As linguagens de domínio específico [2] (DSL) são linguagens de programação, adaptadas a um domínio de problemas [3], a uma técnica de representação ou a uma solução particular. Pretendem restringir as possibilidades dos programadores para o meio para qual foram desenvolvidas. Alguns exemplos da sua aplicação são a programação comportamento de robôs, definição de ambientes gráficos ou até simulações físicas.

Algumas ferramentas de desenvolvimento Web, como é o caso da *Agile Platform*, podem ter como base uma DSL. Estas linguagens, através de construções simples, vão interagir com os diversos componentes do sistema facilitando a comunicação com os repositórios de dados (normalmente definidos em SQL), a manipulação desses dados e sua interface com o utilizador.

Os elevados níveis de abstracção disponibilizados aos programadores tipicamente resultam em perdas de eficiência por parte da linguagem. O código gerado pelo compilador muitas vezes não é optimizado, introduzindo *overheads* significativos resultantes da má estruturação, redundância e excesso de código gerado. Estes problemas geralmente só se começam a manifestar quando as aplicações atingem dimensões consideráveis. É, no entanto, fundamental que eles sejam resolvidos automaticamente durante todo o processo de crescimento da aplicação para evitar modificar todo o código produzido até essa altura.

Neste contexto torna-se essencial proceder a optimizações ao nível do código gerado pelo compilador. No caso particular das *queries*, apesar de existir um meio favorável a optimizações colectivas ao nível dos planos de execução, não se pretende seguir esta abordagem.

A principal motivação incide na existência de padrões de código sub-optimizados que são comuns nos programas realizados pelos diversos programadores da linguagem. Esses padrões surgem não só da relação entre diversas *queries*, mas também da coexistência entre as operações SQL e primitivas típicas de linguagens imperativas, como decisões, ciclos e afectações. O seu aparecimento é muitas vezes resultado tanto da falta de expressividade da linguagem ou elevado nível abstracção, como do estilo ou organização de código *standard* por parte dos programadores.

Esses fragmentos de código podem ser optimizados recorrendo a técnicas de reestruturação do código gerado onde, por exemplo, algumas *queries* podem ser agrupadas, reescritas, eliminadas ou movimentadas.

1.2 Sumário do Problema e Solução Proposta

Após a análise de diversas aplicações foram identificados 6 padrões de código frequentes que são passíveis de ser optimizados. O agrupamento de *queries* foi um dos principais focos do trabalho. Foram estudados casos de consultas que podem ser agrupadas de modo a diminuir a latência das comunicações, aumentar eficiência de ciclos (pre-calculando todos os resultados) ou união de consultas para recolha de resultados em conjunto.

Embora fosse interessante testar os resultados da optimização de todos estes padrões, devido

a limitações de tempo, foi apenas possível implementar por completo a solução para um deles e proceder à devida validação dos resultados. Mas, em todos os restantes casos, foram definidas as limitações actuais e proposta uma forma de as resolver.

O padrão escolhido para implementação baseia-se na necessidade de introduzir operações de actualização parcial para as entidades. Actualmente estas operações necessitam que sejam enviados todos os atributos da entidade no momento da actualização, mesmo que apenas alguns destes tenham sido alterados. Desta forma, qualquer operação de actualização de entidades, obriga a que todos os atributos sejam lidos da base de dados, e em muitos casos desnecessariamente.

A solução proposta consiste na propagação, em tempo de compilação, da informação de quais os atributos que foram modificados ao longo do grafo até à operação de actualização correspondente. Assim, estas operações terão informação suficiente para seleccionar os valores que necessitam de enviar e as *queries* precisarão de recolher menos atributos da base de dados. Por cima desta optimização, será implementado um mecanismo de verificação em tempo de execução, que permite avaliar se os valores realmente alteraram de valor. Este mecanismo irá reduzir o número de atributos enviados nas actualizações, e permitir que nos formulários de edição só sejam actualizados os valores que o utilizador realmente mudou.

1.3 Objectivos e Metodologias

As alterações propostas a estes padrões sub-optimizados serão essencialmente adaptações de técnicas gerais de optimização de compiladores. Envolvem a aplicação de conceitos simples como reposicionamento de código invariante, propagação de valores, restruturação de código e antecipação ou atraso da avaliação de expressões, embora tendo em conta novos cenários.

Para aplicar estes conceitos, muitas vezes é necessário recorrer a técnicas de análise de fluxo de dados para propagar informação ao longo dos grafos. Essa informação pode ser essencial também para calcular dependências entre os diferentes nós. Ao analisar estaticamente o fluxo de informação de cada aplicação, consegue-se prever o conjunto de dados necessário em cada operação e assim reduzir ao essencial a informação que percorre os programas em tempo de execução.

No padrão das actualizações parciais é necessário propagar a informação dos atributos afectados até às operações de actualização das entidades. No fim, essas operações terão informação suficiente para eliminar os atributos desnecessários e deixar de propagar o seu uso até às *queries* que os recolhem da base de dados, optimizando-as e reduzindo o fluxo total de dados em tempo de execução.

Com este trabalho espera-se também que o restante conjunto de alterações propostas permita introduzir melhorias significativas nas aplicações. Estes efeitos serão essencialmente notados ao nível do tempo de execução dos programas. Estas alterações devem reduzir substancialmente o número de ligações aos servidores, retirando algum *overhead* introduzido pelo número actual de pedidos e diminuindo o efeito de engarrafamento (*bottleneck*).

O agrupamento de *queries* deve proporcionar também um melhor uso da *cache* do lado do servidor aumentando a performance do sistema pela diminuição do tempo da sua execução.

A execução destes algoritmos terá sempre um efeito negativo na plataforma porque vai aumentar o tempo de compilação dos programas. Em cada um destes casos haverá a necessidade de avaliar esse custo e decidir se a sua implementação é vantajosa.

1.4 Contribuições do Trabalho

As contribuições desta dissertação são as seguintes:

1) Identificação de padrões gerais de optimização em linguagens com SQL embutido, proposta de soluções de optimização para esses padrões e análise dos seus efeitos semânticos na linguagem.

   Usando a DSL *OutSystems* como caso de estudo, identificámos padrões de código frequentes que são susceptíveis de optimização usando uma série de técnicas desenvolvidas. Para cada um dos padrões identificados, estudámos uma solução adequada (em termos gerais) e determinámos em que medida as optimizações sugeridas poderiam (ou não) alterar a semântica da linguagem e / ou as expectativas do programador. Esta análise foi conduzida com grande detalhe no caso do padrão de foco "Operações de Actualização Parcial de Entidades", onde chegamos a realizar inquéritos sobre as expectativas dos utilizadores. Para os restante casos analisados, esperamos que a informação que produzimos possa facilitar muito a sua análise e implementação mais detalhada no futuro. (Secção 3.4)

2) Introdução de um algoritmo geral de análise estática, baseada em análise de fluxo de dados em operações de actualização parcial de entidades, com o objectivo de diminuir a quantidade de informação manipulada pelos programas. Para o efeito, foi desenvolvida uma extensão ao algoritmo usual de análise de fluxo de dados (*data flow analysis*), e uma adaptação de um algoritmo de análise de actividade (*liveness*) para tirar partido da informação de fluxo de dados. Este algoritmo geral foi completamente implementado na versão real da plataforma *OutSystems*, devendo vir a ser incluido na nova distribuição (versão 5.0), o que demonstra a robustez e qualidade dos resultado obtido. (Secção 4.2.2.1)

3) Análise dos ganhos em desempenho obtidos pela solução proposta, incluindo o seu efeito em aplicações reais de grande dimensão, programadas na plataforma *OutSystems*. Foi realizado um estudo extensivo, documentado em várias tabelas comparativas, referindo uso de recursos (espaço e tempo). (Secção 4.2.3)

4) Definição de um mecanismo adicional de optimização, baseado na combinação de informação estática (gerada pela análise estática) com informação dinâmica (obtida em tempo de execução), para melhorar ainda mais os benefícios do nosso algoritmo de optimização. (Secção 4.2.2.2)

1.5 Estrutura do Documento

Este documento está organizado da seguinte forma:

- O segundo capítulo faz a ligação dos objectivos desta dissertação com o trabalho relacionado e estado da arte. São apresentados trabalhos e técnicas para optimização de múltiplas *queries*, descrito o funcionamento de um optimizador e apresentadas técnicas para a optimização de compiladores relevantes para este trabalho.
- O terceiro capítulo aborda o enquadramento da dissertação e descreve a arquitectura e funcionamento da *Agile Platform*, dando especial relevo à linguagem que vai ser optimizada. São explicadas as construções mais relevantes, os diversos tipos de dados e a estrutura do modelo das base de dados. Por fim, são identificados os padrões sub-optimizados e descritas as respectivas optimizações propostas, dando especial ênfase aquela que foi implementada.
- O quarto capítulo introduz o restante trabalho realizado no âmbito desta dissertação. Começa por mostrar de forma breve a arquitectura do compilador *OutSytems* para melhor compreensão do enquadramento destas optimizações. Depois é descrito o trabalho de estudo, implementação e validação realizado na introdução das operações de actualização parcial na linguagem.
- Finalmente, a última secção do documento faz o levantamento das conclusões finais sobre o trabalho realizado e daquilo que pode ser futuramente melhorado e acrescentado.

2

Técnicas de Optimização de Código

Antes de prosseguir para a resolução do problema descrito anteriormente, tornou-se fundamental rever um conjunto de metodologias e trabalhos relacionados com a temática desta dissertação. Neste capítulo vão ser abordadas duas classes distintas de técnicas. Na primeira secção, são apresentadas algumas técnicas de optimização de compiladores que englobam mecanismos de reordenação, factorização de código, análise de fluxo de dados e eliminação de subexpressões comuns. Numa segunda fase, é abordado o tema da optimização de *queries* SQL e mostrada uma visão desde a arquitectura dos optimizadores dos SGBD actuais, até aos métodos utilizados para optimizar aplicações que manipulam conjuntos de *queries*.

A escolha destes temas pretendeu abranger ao máximo as dificuldades que pudessem surgir na elaboração deste trabalho. As técnicas de optimização de compiladores estudadas terão uma aplicação muitas vezes directa neste trabalho. Por outro lado, os conceitos abordados na parte do SQL permitiram ter noção das barreiras existentes na manipulação de múltiplas *queries* e de algumas técnicas existentes para as contornar, e apesar de tudo não foram tão importantes no decorrer deste trabalho.

Na conclusão deste capítulo são analisados os aspectos mais importantes destes trabalhos e das técnicas descritas, fazendo um paralelismo com o trabalho desenvolvido.

2.1 Transformação e Optimização de Programas

A optimização de compiladores é o processo de transformação de código que permite maximizar a performance dos programas gerados. As optimizações mais habituais englobam técnicas para reduzir o tempo de execução de programas e a utilização de memória. A principal

premissa de qualquer optimização é que estas nunca vão alterar a correcção dos programas.

Algumas destas optimizações resultam em problemas de elevada complexidade computacional, que podem elevar muito o tempo de compilação das aplicações, tonando-se indesejáveis para os programadores devido à alta relação custo benefício.

Esta secção descreve algumas técnicas de optimização de compiladores, organizando-as em três subsecções. Na primeira, são enumeradas técnicas gerais de análise de fluxo para suporte à reordenação e remoção de código. Aqui é dada especial importância ao controlo de dependências e às técnicas de análise de fluxo de dados para propagação de informação nos grafos dos programas. Na secção seguinte é apresentado o problema da identificação e eliminação de subexpressões comuns. Por último, são discutidas algumas técnicas mais específicas de factorização de código, que englobam propagação de valores, antecipação de expressões ou movimentação de código invariante em ciclos.

2.1.1 Análise de Fluxo

As técnicas de reorganização global do código dos programas podem não ser directamente responsáveis por melhorias na eficiência. Contudo, a sua aplicação poderá fornecer condições para que outras optimizações actuem de forma mais eficaz.

Ao analisar o fluxo de controlo e de dados recolhemos informação adicional sobre o comportamento dos programas que é bastante rica para a aplicação algoritmos de optimização. Por esta análise observamos que podem existir zonas de código que nunca serão executadas ou que são executadas em ciclos. Contudo, se precisarmos de proceder a movimentação de código, é fundamental verificar as dependências entre os conjuntos de dados manipulados em cada ponto do programa para que a semântica dos programas não se altere. Essas dependências podem ser calculadas utilizando técnicas de análise do fluxo de dados.

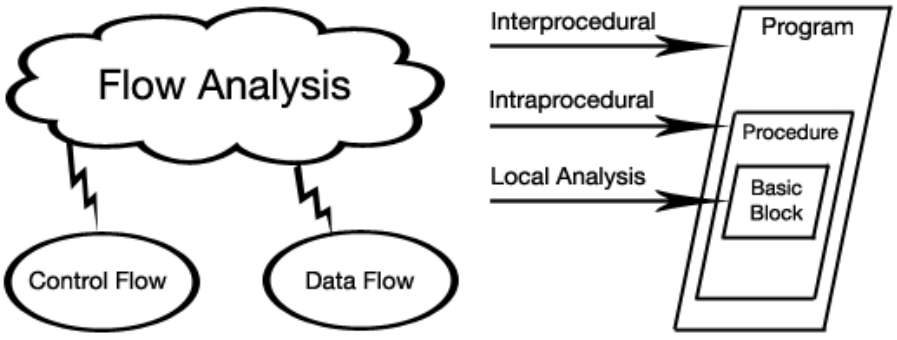

Figura 2.1: Tipos de análise de fluxo

Os dois tipos de análise de fluxo referidos podem ser realizados ao nível dos programas (Inter-procedimental), de cada procedimento (Intra-procedimental) ou de cada bloco básico (local). A figura 2.1 mostra esquematicamente esta hierarquia.

2.1.1.1 Análise do Fluxo de Controlo

Na generalidade, quando um programa é lido pelo compilador, passa por um conjunto de operações de análise gramatical sobre o código lido (*parsing*) até chegar a um resultado final, que tipicamente é uma árvore sintaxe abstracta (AST). Em algum momento, durante ou após este processo, o compilador tem hipótese de recolher informação sobre o programa que vai executar, e como é que essa execução vai ser feita.

O processamento da informação de um programa, de forma a obter este conjunto de dados, chama-se análise de fluxo de controlo [4]. O primeiro passo desta análise passa por construir uma estrutura que separa o programa alvo em blocos básicos de controlo, ligando-os sob a forma de grafo. Os blocos básicos são parcelas de código consecutivas, em que o fluxo de controlo apenas pode aceder no início e deixar no seu fim (sem *jumps* intermédios). As arestas desse grafo representam o fluxo de controlo e a estrutura final tem o nome de grafo de controlo de fluxo (CFG - *Control Flow Graph*). Os CFG abstraem a estrutura do programa original num grafo finito e único para todas as execuções.

Esta análise permite obter informação sobre zonas de código que nunca são alcançadas, através da simples verificação da existência de nós sem ligações. Cada aresta significa que o programa pode seguir aquela direcção em tempo de execução. Isto é particularmente útil para avisar do programador de código que nunca vai ser executado e proceder à sua remoção.

A análise de fluxo de controlo é a base para muitas outras análises estáticas de código, podendo ser melhorada se acrescentarmos informação dos dados que são enviados entre os blocos básicos. Essa tarefa cabe à análise de fluxo de dados, que é apresentada na subsecção seguinte.

2.1.1.2 Análise de Fluxo de Dados

Para as optimizações conseguirem tirar total partido da informação manipulada, o compilador precisa de olhar para o conjunto de dados como um todo e distribuir essa informação pelos diversos nós do grafo de fluxo. A análise de fluxo de dados [5] é a técnica utilizada para propagar pelos diversos nós, a informação sobre os dados que estão a ser manipulados.

Os grafos de fluxo de dados (DFG - *Data Flow Graph*) representam as dependências de dados entre as diversas instruções dos programas. Cada nó (S) do grafo representa um conjunto de instruções (um bloco básico ou instruções singulares) e contém uma porta de entrada de informação (in[S]) e outra de saída (out[S]). A informação que entra em cada nó em in[S], é manipulada e depois propagada em out[S]. Tipicamente essa manipulação pode ser descrita por uma função, da seguinte forma:

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

onde cada tipo de nó define as suas funções *gen*[S] e *kill*[S]. A equação diz que cada nó pode eliminar (*kill*) um subconjunto de elementos do conjunto de entrada e acrescentar nova informação (*gen*) a esse conjunto.

Quando a propagação se dá no sentido inverso à execução do programa (*backward flow analysis*) usualmente usa-se a seguinte notação:

$$in[S] = gen[S] \cup (out[S] - kill[S])$$

que tem o mesmo comportamento, visto que apenas se dá uma troca entre os canais de entrada e de saída.

Para além disto, cada um dos nós geralmente tem associadas duas listas: *def* e *use*. A primeira representa o conjunto de definições de identificadores geradas nesse nó e a segunda a lista de identificadores lidos.

De uma forma geral, qualquer tipo de propagação define para cada nó do DFG duas novas listas de identificadores: *in*[S] e *out*[S] e duas funções *kill*[S] e *gen*[S]. A próxima subsecção descreve um algoritmo para propagação desta informação.

2.1.1.3 Método do Ponto Fixo

Num conjunto S de ordem parcial $\sqsubseteq$, são respeitadas as seguintes propriedades:

- $\sqsubseteq$ é uma relação em S
- Reflexividade: $\forall_a \in S : a \sqsubseteq a$
- Transitividade: $\forall_{a,b,c} \in S : a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$
- Anti-simetria: $\forall_{a,b} \in S : a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$

Esse conjunto é considerado um reticulado (*Lattice*) se para cada subconjunto não vazio C de S for possível definir:

- α como GLB (*Greatest Lower Bound*) de C

$$\cdot \forall_x \in C : \alpha \leq x$$
$$\cdot \forall_{k=LB(S)} \Rightarrow k \leq \alpha$$

- β como LUB (*Least Upper Bound*) de C

$$\cdot \forall_x \in C : \beta \geq x$$
$$\cdot \forall_{k=UB(S)} \Rightarrow k \geq \beta$$

Se $(S, \sqsubseteq)$ for um qualquer reticulado completo e $f : S \to S$ uma função monótona crescente, ou seja, $\forall_{x,y} \in S : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$, então segundo Tarsky [6], o conjunto de todos os pontos fixos de f são um reticulado completo respeitando $\sqsubseteq$.

Isto implica que qualquer função monótona f num reticulado R, tem um único ponto fixo mínimo, ou seja, um valor mínimo $x \in R : f(x) = x$, e analogamente um único ponto fixo máximo.

O conjunto de informação propagada num DFG corresponde a um reticulado R de tamanho finito. Quando o grafo tem n nós, qualquer função monótona $f : R^n \to R^n$ que defina as regras de propagação de um conjunto de nós, pode gerar um reticulado $x \in R^n : x = f(x)$.

Se todas as condições acima estiverem reunidas, o ponto fixo máximo pode ser calculado pelo algoritmo iterativo 2.1.1.

Algorithm 2.1.1: PONTO FIXO MÁXIMO(*pseudocodigo*)

$r \leftarrow \{\}$
do $\begin{cases} copy \leftarrow r \\ r \leftarrow f(r) \end{cases}$
while $copy! = r$
return (r)

2.1.1.4 Ordenação Topológica

A ordem pela qual são percorridos os nós do grafo pode ter influência no número de iterações realizadas pelo algoritmo iterativo de propagação. A propagação de informação num grafo estabiliza mais rapidamente, se para cada nó de uma determinada iteração, os seus predecessores já tenham propagado a informação nessa mesma iteração. Em [7] é apresentado um algoritmo de ordenação topológica para iteração dos nós de um grafo orientado acíclico.

A ordenação topológica dos nós um grafo, pode ser vista como a disposição de todos esses nós sobre uma linha horizontal, onde cada um têm todas as suas ligações direccionadas para nós mais à frente nessa linha.

Uma forma simples de implementar este algoritmo (2.1.2) consiste em percorrer repetidamente uma lista inicialmente preenchida com todos os nós, retirando da lista a cada iteração, o primeiro elemento que não tem ligações de entrada (actualizando as entradas dos nós de destino) até a lista inicial ficar vazia. A complexidade deste algoritmo é no entanto bastante

elevada: $O(n^3)$.

Algorithm 2.1.2: ORDENAÇÃO TOPOLÓGICA(*pseudocodigo*)

$$
\begin{array}{l}
ordered \leftarrow \{\} \\
all \leftarrow \{Node_1, ..., Node_n\} \\
\textbf{while } all! = \{\} \\
\quad \textbf{do} \left\{ \begin{array}{l} \textbf{for each } node \in all \\ \textbf{if } node.predecessors = \{\} \\ \quad \textbf{then} \left\{ \begin{array}{l} \textbf{for each } succ \in node.successors \\ \quad succ.predecessors.remove(node) \\ ordered \leftarrow ordered \cup node \end{array} \right. \end{array} \right. \\
\textbf{return } (ordered)
\end{array}
$$

A implementação mais típica deste algoritmo envolve uma pesquisa em profundidade em grafos (DFS: Depth-First Search [8]). Para além da complexidade mais reduzida (apenas $O(n)$), em termos de implementação é mais fácil a detecção de ciclos. Este último factor é muito importante já que foi provado [9] que só se garante que o número de iterações é mínimo em grafos acíclicos.

Para contornar a existência de ciclos, o método mais utilizado passa pela criação de uma estrutura DFST [4] (Dept-First Spannig Tree) categorizando as arestas como aresta de avanço, recuo ou neutra (nenhuma das anteriores).

A implementação actual do compilador *OutSystems* realiza uma ordenação topológica, baseada no conceito *best effort* (melhor ordenação possível), uma vez que os grafos são quase sempre cíclicos.

2.1.1.5 Análise de Variáveis Activas

O método para verificação de variáveis activas (*Liveness Analysis*) num programa [10] é um caso clássico de análise de fluxo de dados. Uma variável é considerada activa (*live*) à saída de um nó, se existir a hipótese do seu valor vir a ser utilizado mais à frente na execução do programa.

A solução para este problema consiste na propagação no grafo de fluxo, no sentido inverso ao da execução, o conjunto de identificadores lidos em cada nó, aplicando as seguintes regras.

$$Live_{in}[S] = gen[S] \cup (Live_{out}[S] - kill[S])$$

$$Live_{out}[S : endnode] = \emptyset$$
$$Live_{out}[S] = \cup_{n \in Successors[S]} Live_{in}[n]$$

$$gen[S : y \leftarrow f(x_1, ..., x_n)] = \{x_1, ..., x_n\}$$
$$gen[S : others] = \emptyset$$
$$kill[S : y \leftarrow f(x_1, ..., x_n)] = \{y\}$$
$$kill[S : others] = \emptyset$$

Actualmente, o compilador *OutSystems* possui uma análise de variáveis activas para suportar optimizações ao nível da informação que precisa de ser lida da base de dados no contexto das *Simple Queries*. A mesma implementação permite optimizar a informação que acompanha a navegação entre páginas e é apelidada internamente de *Viewstate optimization*.

Depois da execução do algoritmo, as *queries* só precisam de recolher da base de dados os atributos que estiverem marcados como *live* à saída desse nó. No caso da *Viewstate optimization*, só precisam de ser gravados no *Viewstate* os valores que estiverem marcados como *live* à saída do nó de transição de página. Posteriormente, esses valores não vão ser lidos, ou então marcados com o valor por defeito.

2.1.1.6 Expressões Muito Activas

Uma expressão é denotada como muito activa [9] quando num determinado ponto do grafo do programa, qualquer que seja o caminho seguido na execução, essa expressão vai ser avaliada antes de que qualquer valor dos seus operandos se altere. Como isso implica ter conhecimento do futuro de cada uma destas expressões, o percurso tem que ser feito no sentido inverso (backward flow analysis).

Esta análise de expressões é também ideal para movimentação de código invariante em ciclos (ver 2.1.3.1). As expressões que são invariantes em ciclos são ao mesmo tempo muito activas, pois sabemos que vão ser usadas no futuro e o seu valor não vai ser alterado. Assim, podem ser transportadas e avaliadas fora do ciclo.

Este algoritmo associa a cada nó do grafo, um conjunto de expressões consideradas muito activas nesse ponto. Cada expressão do programa pode aparecer repetida em vários nós do grafo, sendo que esses pontos são aqueles onde a sua computação terá sempre o mesmo resultado. O conjunto de regras a aplicar com o método do ponto fixo são [9] as seguintes.

$$Busy_{in}[S] = gen[S] \cup (Busy_{out}[S] - kill[S])$$

$$Busy_{out}[S : endnode] = \emptyset$$
$$Busy_{out}[S] = \cap_{\, n \in Successors[S]} Busy_{in}[S]$$

$$gen[S] = \cup_{\, e \text{ is expression} \in S} \{e\}$$
$$kill[S : x \leftarrow E] = \cup_{\, e \in Busy_{out}[S] \ \& \ e \text{ uses } x} \{e\}$$
$$kill[S : others] = \emptyset$$

Este algoritmo poderá ser integrado no compilador com o objectivo de encontrar para cada *query*, os pontos do grafo para qual a sua execução tem o mesmo resultado. Sabendo isto, podemos identificar zonas de aglomerados de *queries* e fazer a movimentação de algumas destas, se o efeito for benéfico. Esta transformação está explicada em mais detalhe em 3.4.1.1.

2.1.2 Eliminação de Subexpressões Comuns

A eliminação de subexpressões comuns [4] é uma técnica bastante comum em optimização de compiladores. As expressões podem ser vistas também elas como um conjunto de expressões, denominadas subexpressões. Algumas delas podem ser partilhadas por um conjunto de operações, como que um padrão que se repete em vários locais no código. Se o valor dessas subexpressões for o mesmo, ou seja, se os valores de variáveis utilizadas não forem alterados entre as expressões, elas podem ser avaliadas uma única vez. O valor seria guardado numa variável e o seu resultado utilizado em todos os locais onde o padrão aparece.

Os benefícios desta abordagem são evidentes. Como a expressão só é avaliada uma vez, o ganho vai ser directamente proporcional ao número de expressões que conseguimos eliminar e o seu impacto varia consoante o seu grau de complexidade.

Esta técnica requer que seja criada uma variável adicional por cada subexpressão pré-calculada e a criação excessiva de variáveis pode ter um impacto negativo na performance dos programas. Desta forma é importante que a escolha das expressões a guardar em variáveis temporárias seja cuidada, isto é, em casos onde o impacto na performance seja realmente evidente.

Uma aplicação muito simples deste conceito é mostrada no seguinte exemplo:

$$x \leftarrow a * b^{10} + v$$
$$y \leftarrow a * b^{10} * c$$

que seria transformado em :

$$aux \leftarrow a * b^{10}$$
$$x \leftarrow aux + v$$
$$y \leftarrow aux * c$$

Quando utilizamos esta técnica em conjuntos de *queries*, a partilha de valores não é tão trivial, e nem sempre tem um impacto positivo nos programas. A secção 2.2.2.1 apresenta alguns exemplos da sua aplicação e é inclusivamente apresentado um exemplo onde a aplicação pode ser prejudicial.

2.1.3 Factorização

Em certos casos, a movimentação de expressões para zonas diferentes do código pode ter um impacto forte no desempenho dos programas. Um exemplo clássico é a movimentação de código invariante em ciclos 2.1.3.1, onde o cálculo das mesmas expressões se repete, quando poderiam ser calculadas apenas uma vez antes do ciclo.

Outros casos mais sofisticados podem envolver controlo e partilha de dados em memória. Algumas expressões podem ter ganhos quando executadas em série, mesmo que sejam completamente independentes se, por exemplo, precisarem que os mesmo dados estejam carregados em memória.

Explorar o carregamento dos dados pode ter efeitos muito benéficos para as aplicações. O principio da localidade [11], como é denominado, pode ser explorado quando o comportamento dos programas é previsível.

Uma aplicação clássica deste princípio consiste na comutação das iterações dos cilos (*loop interchange*) no acesso a elementos de um vector de múltiplas dimensões. Se os elementos forem acedidos linearmente a eficiência vai ser maior do que se o acesso for transversal, porque há menos riscos de ser necessário carregar de novo as páginas para memória (menos *page faults*).

Exemplo de comutação de ciclos:

$$\begin{array}{l}\textbf{for } j \leftarrow 0 \textbf{ to } 100\\ \quad \textbf{for } i \leftarrow 0 \textbf{ to } 200\\ \quad\quad a[i,j] \leftarrow i+j\end{array}$$

depois da devida transformação ficaria:

$$\begin{array}{l}\textbf{for } i \leftarrow 0 \textbf{ to } 200\\ \quad \textbf{for } j \leftarrow 0 \textbf{ to } 100\\ \quad\quad a[i,j] \leftarrow i+j\end{array}$$

2.1.3.1 Movimentação de Código Invariante em Ciclos

Esta operação, também apelidada de *Hoisting* [4, 10], consiste na detecção e antecipação de código invariante nos ciclos dos programas. Esta movimentação permite que algum código, que antes era repetido durante todo o ciclo, seja calculado apenas uma vez. Os ganhos desta operação dependem naturalmente da complexidade das expressões envolvidas e da dimensão dos ciclos.

Uma definição ($d : t \leftarrow a_1\ op\ a_2$) é invariante num ciclo se para cada um dos seus operandos (a_i) se verificarem as seguintes propriedades [4, 10]:

i. a_i é constante

ii. ou, todas as definições a_i que alcançam d estão fora do ciclo

iii. ou, há exactamente uma definição de a_i que alcança d, e essa definição é invariante do ciclo.

Uma aplicação muito simples deste conceito é mostrada no seguinte exemplo:

$$
\begin{array}{l}
i \leftarrow 0 \\
\textbf{while } i++ < (10+x) \\
\quad array[i] \leftarrow x^3
\end{array}
$$

que seria transformado em:

$$
\begin{array}{l}
i \leftarrow 0 \\
cube \leftarrow x^3 \\
limit \leftarrow 10+x \\
\textbf{while } i++ < limit \\
\quad array[i] \leftarrow cube
\end{array}
$$

Esta técnica será utilizada no caso das *queries* que se repetem constantemente em ciclos. O problema é que, devido à concorrência entre processos e ao baixo nível de isolamento utilizado na ferramenta (*read uncommited* - em SQL Server), as *queries* não são invariantes nos ciclos. Aplicar este método implicaria alterar a semântica dos programas. Na secção 3.4.1.5 são debatidas as implicações e os detalhes desta optimização.

2.1.3.2 Propagação por Cópia

Esta operação, descrita em [4], consiste na eliminação de uma variável que apenas servia para guardar temporariamente o resultado de uma expressão, substituindo o uso dessa variável no resto do programa pela própria expressão. Isto implica verificar se a variável não é mais utilizada no programa e esta transformação pode não ser benéfica se implicar avaliar a expressão mais vezes. Uma aplicação muito simples deste conceito é mostrada no seguinte exemplo:

$$
\begin{array}{l}
y \leftarrow x \\
z \leftarrow 3+y
\end{array}
$$

que seria transformado em:

$$z \leftarrow 3+x$$

Esta técnica quando aplicada sobre *queries* pode ter um impacto forte. À partida, observa-se que é eliminada uma das *queries*, o que significa que temos menos ligações à base de dados. Outro factor importante é que o optimizador de *queries* encarrega-se de calcular a melhor forma para realizar a operação. A secção 3.4.1.2 mostra um exemplo da aplicação directa desta técnica em pares de *queries* dependentes.

2.2 SQL e Bases de Dados Relacionais

A versão do SQL original chamava-se *Sequel* 2 e foi apresenta por Chamberlin et al. em [12] como uma adaptação das linguagens *Square* [13] e *Sequel* [14]. O nome foi mais tarde alterado para SQL (como acrónimo para *Structured Query Language*) e com o passar dos anos esta linguagem tornou-se o standard para as bases de dados relacionais. O SQL é uma linguagem declarativa de pesquisa em bases de dados que assenta sobre as operações da álgebra relacional (selecção, projecção, renomeação, produto cartesiano, união e diferença de conjuntos).

Uma base de dados relacional [15] consiste num conjunto de tabelas identificadas por um nome único, onde cada linha dessas tabelas define uma relação entre o conjunto de valores. Informalmente, uma tabela constitui um conjunto de entidades e cada linha dessa tabela representa uma entidade.

Os sistemas de gestão de bases de dados (SGBD) fornecem suporte para a criação, consulta e alteração de valores destas bases de dados. Os comandos SQL enviados são interpretados internamente e traduzidos em operações relacionais sobre os dados. Como a eficiência das operações combinatórias da álgebra relacional dependem muito do estado interno das tabelas, e também para facilitar o uso da linguagem, os SGBD possuem implementado um optimizador de planos de acesso aos dados.

A secção seguinte descreve o funcionamento desse optimizador.

2.2.1 Optimização de *Queries*

Antes de proceder à análise das optimizações de múltiplas *queries* é necessário conhecer bem o que acontece quando apenas uma *query* é submetida ao SGBD. Este processo foi descrito detalhadamente nos trabalhos de Ioannidis [16] onde são enumeradas de uma forma abstracta todas as fases por onde uma *query* passa após ser submetida. É natural que, dependendo da implementação da plataforma, os módulos seguintes tenham uma divisão mais ou menos clara mas acabam por estar todos presentes (com excepção do *Rewriter*).

A imagem anterior mostra que a arquitectura divide os módulos em duas etapas diferentes. A primeira é a declarativa e é constituída apenas por um módulo de rescrita, responsável por remodelar cada *query* submetida, de forma a criar outras *queries* equivalentes e potencialmente mais eficientes. As transformações realizadas nesta etapa englobam, entre outras, transformações de sub-*queries* em operações de junção, rescrita de *queries* para fazer uso de vistas materializadas e separação de *queries* aninhadas.

Estas optimizações são consideradas avançadas e estão apenas implementadas em alguns SGBD, como é o caso do *DB*2, *Oracle* e *Illustra*.

A segunda etapa (procedimental) vai considerar todas as *queries* elaboradas na fase anterior e escolher aquela que tem os custos mais baixos para ser executada. A maior parte desse processo é efectuado no planeador que vai recorrer aos módulos auxiliares desta etapa para escolher o plano mais adequado.

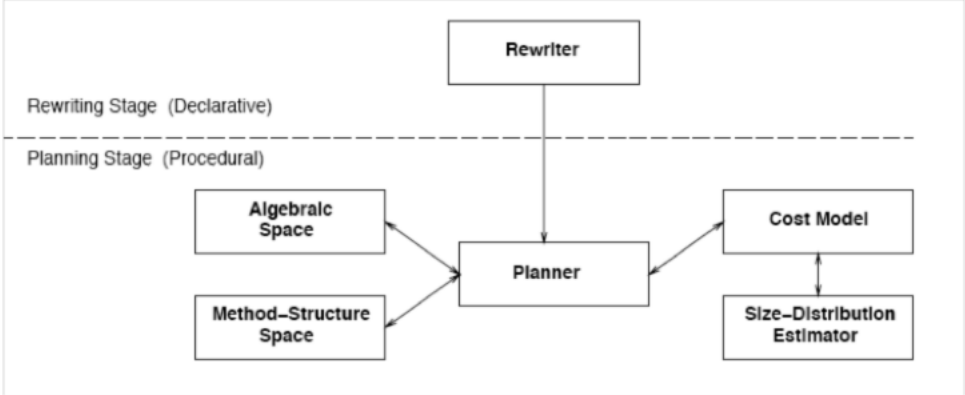

Figura 2.2: Arquitectura do optimizador de *queries*

No módulo *Algebraic Space* são examinadas as árvores de cada *query*, normalmente representadas em álgebra relacional, e determinada a melhor ordem das suas operações internas. É importante realçar que qualquer que seja a ordem definida, o resultado final será sempre o mesmo, uma vez que todas as árvores analisadas são equivalentes. Depois de definida a ordem das operações, é necessário decidir quais os algoritmos a utilizar no processamento da *query*. Esta escolha é feita no módulo *Method-Structure Space*, onde são seleccionados os algoritmos de junção, definido o uso de índices de determinadas tabelas e outras características que podem depender também muito das propriedades do SGBD.

Para o planeador poder escolher entre os diferentes planos de execução, tem que existir uma forma de estimar quais os custos de cada plano, de acordo com as operações efectuadas por cada um. Para tal existe o *Cost Model*, que disponibiliza fórmulas aritméticas que aproximam os valores dos custos das operações de controlo de *buffers*, CPU, I/O, etc. Estas fórmulas precisam naturalmente dos valores do tamanho das tabelas e índices para poderem ser calculadas. Esses valores são dados pelo último módulo: *Size-Distribution Estimator*.

2.2.2 Optimização de Múltiplas *Queries* (MQO)

A optimização de uma única *query*, como foi descrita anteriormente, é um processo complicado que envolve vários passos intermédios. Estes algoritmos são pesados porque, tipicamente, envolvem testar todas as representações possíveis de uma *query* e estimar para cada uma delas a sua eficiência. Este procedimento pode ser melhorado dado que algumas operações equivalentes são sempre mais eficientes que outras, podendo poupar alguns cálculos desnecessário. Alguns SGBD permitem também fazer *cache* destes planos de acesso permitindo assim, para cada *query*, guardar o seu plano de execução para o caso desta vir a ser chamada num futuro próximo. No entanto, os seus ganhos só se reflectem após cerca de 3 ou 4 chamadas da mesma *query*.

Actualmente, grande parte dos SGBD já têm também algum suporte para a optimização de múltiplas *queries*, mas fazem-no apenas no contexto das *queries* aninhadas (*nested queries*). No âmbito deste trabalho, o principal foco de importância são os grupos de *queries* que são independentes, e fundamentalmente perceber e tentar explorar os seus interesses comuns.

A construção de planos de acesso optimizados para múltiplas *queries* é um problema NP-Difícil, como foi provado por Sellis e Ghosh em [17]. O plano ideal para um conjunto de *queries*, não é necessariamente a conjunção de todos os melhores planos de acesso individuais. Operações como partilha de subexpressões comuns, podem melhorar muito o desempenho e envolvem ao mesmo tempo alterar os planos óptimos de cada uma.

Em [18], Sellis et al. formalizaram o problema da optimização de múltiplas *queries*. Nele definiram uma base de dados como um conjunto de relações $R = \{R_1, R_2, \cdots, R_n\}$, cada uma definida por um conjunto de atributos; e uma *query* Q como uma sequência de tarefas ou operações que produzem um conjunto de resultados. O processamento de uma *query* envolve tanto utilização de CPU como de I/O, que definem o custo da operação.

Dado um conjunto de *queries* $Q = \{Q_1, Q_2, \cdots, Q_n\}$, um plano de acesso global consiste numa forma de computar o resultado de todas as *queries*. Uma forma simples de o fazer será juntar todos os melhores planos de acesso individuais de cada *query* Q_i.

Foram definidos dois conceitos: *locally optimal*, para definir o melhor plano de acesso para cada *query* individual e *global optimal* (GP), para definir o plano de acesso global com o menor custo possível. O problema da optimização de múltiplas *queries* ficou assim definido:

- Dado um conjunto de n planos de acesso $P = \{P_1, P_2, \cdots, P_n\}$, com $P_i = \{P_{i_1}, P_{i_2}, \cdots, P_{i_{ki}}\}$, e sendo P_i o conjunto de todos os planos possíveis para processar $Q_i, 1 \leq i \leq n$,

- Encontrar um plano de acesso global GP, seleccionando um plano de cada P_i de forma a que GP seja mínimo.

Nesta secção serão descritas algumas técnicas utilizadas em trabalhos onde foi necessário introduzir optimizações ao nível do processamento de múltiplas *queries*. As abordagens seguidas neste contexto envolvem, entre outras, técnicas para reduzir os custos da detecção de subexpressões comuns [19, 20, 21]; implementação de camadas ou extensões aos optimizadores de *queries* actuais, para tratamento de conjuntos de *queries* [22, 17, 18, 23, 24]; ou até mesmo implementar um optimizador de múltiplas *queries* de raiz [25], para actuar antes destas serem enviadas para o SGBD.

2.2.2.1 Isolamento de Subexpressões Comuns

Uma subexpressão é uma fracção de uma *query* que define um resultado intermédio que é usado durante o processo de avaliação dessa *query*. Alguns desses resultados intermédios podem ser iguais em conjuntos de *queries* que são enviadas à mesma altura para o SGBD, podendo naturalmente ser partilhados entre todas.

O problema de identificar subexpressões comuns em conjuntos de *queries* é complexo. Várias abordagens foram seguidas ao longo dos tempos para solucionar este problema mas a complexidade das soluções é um forte entrave à integração destes algoritmos nos SGBD actuais. A verdade também, é que a partilha de operações nem sempre traz vantagens para os planos de acesso, vejamos o exemplo da figura 2.3:

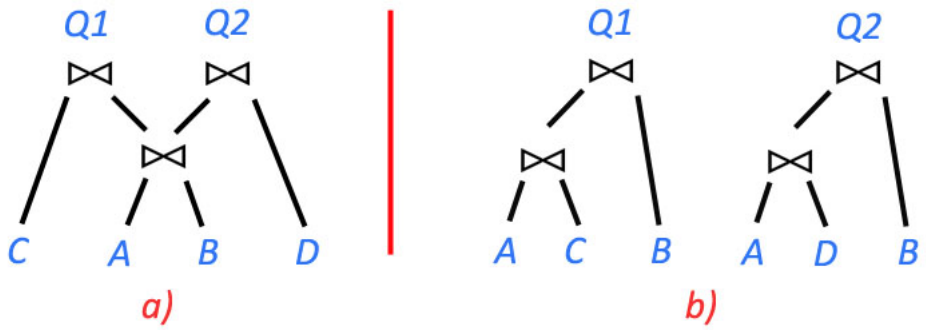

Figura 2.3: Exemplos de planos de execução para duas *queries* (com e sem partilha de resultados)

Na imagem anterior os planos de *a*) parecem ser mais eficientes porque partilham o resultado da junção de *A* com *B*, mas podem na realidade ser muito menos eficientes. Isto pode acontecer quando as tabelas *C* e *D* são muito grandes e a resultante da junção de *A* com *B* também o é. Ao mesmo tempo, em *b*) os resultados das junções de *A* com *C* e de *A* com *D* podem ter resultados muito pequenos e reduzir assim muito o número de combinações com *B*, aumentando a eficiência global do plano.

O reconhecimento de subexpressões em *queries* é um desafio interessante, não só ao nível dos algoritmos utilizados, mas também na forma como são representadas as *queries* para a aplicação dos mesmos. Em [19], as *queries* são representadas em formas de *strings* de símbolos e, posteriormente, é aplicado um algoritmo heurístico de trepa-colinas para encontrar a melhor combinação de subexpressões comuns e eram apenas consideradas *queries* individuais. Mais tarde, em [21], é proposta uma representação das *queries* em forma de grafo (*query graph*). Nos *query graphs* os nós correspondem a ocorrências de tabelas na *query* e os arcos correspondem às junções entre as tabelas nos nós a que ligam. Os arcos não são orientados e são etiquetados com a condição de junção das tabelas. Com base nesta representação, é apresentada uma metodologia para encontrar resultados comuns no processamento de conjuntos de *queries ad hoc*. O processamento do grafo permite a verificação da existência de resultados temporários guardados dos pedidos anteriores. É apresentada também uma framework abstracta para fazer uso desses mesmos resultados.

A construção de um plano global de acessos foi abordada de uma forma diferente em [20].

Esta solução tem como base um algoritmo de programação dinâmica para identificar e seleccionar de entre todos os planos de acesso individuais, o plano global que tenha o custo de execução mínimo. A ideia consiste em olhar para cada *query*, retirar as propriedades dos acessos e, de acordo com essas propriedades, construir o plano global. A selecção desse plano global é realizada sobre um grafo não orientado, onde cada nó representa um plano de acesso P_{ij}, do qual i simboliza a *query* e j um plano dessa mesma *query*. Existe uma métrica associada a cada um desses nós que caracteriza o custo da execução (> 0), e o mesmo também para cada transição ($\leqslant 0$) representando o ganho potencial devido à partilha de tarefas em comum entre os planos. O objectivo do algoritmo é encontrar o plano que execute todas as *queries* i, e que tenha o custo mínimo. É de realçar que apenas alguns dos nós do grafo vão ser seleccionados para serem executados, uma vez que estão representados vários planos para a mesma *query*.

Por último, Jarke em [21] fez uma análise sobre este problema em três linguagens diferentes. Foram analisadas técnicas para reconhecer, suportar e explorar subexpressões comuns em: *Relational Algebra*, *Domain Relational Calculus* (QBE e Prolog) e *Tuple Relational Calculus* (SQL e QUEL). No mesmo trabalho foram discutidas vantagens e desvantagens do envio simultâneo de *queries*, com base nas demonstrações realizadas em [26].

2.2.2.2 Optimizações com Base em Heurísticas

Antes da pesquisa por planos de execução com base em heurísticas ser utilizada no contexto de múltiplas *queries*, já tinham sido introduzidas por Hall em [19] no âmbito de optimizações de expressões singulares. O objectivo deste trabalho era, em vez de tentar encontrar o melhor plano possível de execução, encontrar expressões que são aparentemente mais eficientes. Foram introduzidas tanto equivalências que é sabido serem sempre mais eficazes, como algumas regras *ad hoc* que são geralmente sempre úteis. Os resultados finais foram descritos sobretudo como melhoramentos em vez de optimizações, pois não havia garantias que os planos optimizados tivessem um impacto positivo no resultado final.

A necessidade do uso de heurísticas no contexto da optimização de múltiplas *queries*, surge naturalmente da complexidade do problema. Como já foi dito anteriormente, trata-se de um problema NP-Difícil, e os algoritmos de pesquisa baseados em heurísticas como o A^* [8] permitem reduzir em muito a quantidade de nós explorados, aumentando a eficácia, embora só consigam garantir o melhor plano possível se a heurística usada for admissível [8].

Encontrar uma boa heurística admissível para este problema é uma tarefa extremamente complicada. As soluções com base em heurísticas geralmente não procuram achar o melhor plano possível mas sim encontrar uma boa aproximação, expandindo ao mínimo o espaço de procura.

Em [22], Sellis propõe uma heurística sofisticada (HA) que procura a melhor conjunção de todos os planos de acesso de cada *query*. É feita uma comparação de performance com um algoritmo que tenta entrelaçar os planos de acesso das *queries* (IE), com operações partilhadas. De acordo com os testes realizados, o algoritmo HA mostrou-se, no pior dos casos, igual a IE. Posteriormente no trabalho desenvolvido em [18] são propostos melhoramentos a este

algoritmo, e também a [17], reduzindo o número de nós expandidos. Ambos são bons exemplos do uso de heurísticas em MQO.

Mais recentemente no trabalho [23], foram apresentados três novos algoritmos heurísticos, e demonstrou-se que a sua integração nos optimizadores actuais pode ser feita de uma forma simples. Estes pretendem aproveitar o facto das *queries* poderem partilhar operações e resultados intermédios, que era a maior falha das abordagens anteriores.

Também baseado na ideia de não introduzir grandes alterações no motor da base de dados, [24] apresenta um algoritmo para decomposição de *queries* que torna mais fácil testar as suas equivalências a nível lógico. São introduzidos novos operadores relacionais para capturar a semântica da partilha de resultados. As heurísticas foram utilizadas para cortar o espaço de procura de estratégias de combinação de *queries*.

Uma solução mais sofisticada foi apresentada em [25], onde foi criado um optimizador de *queries* de raiz sobre o optimizador do SGBD. Neste trabalho, procurou-se optimizar os planos de *queries* que contemplam múltiplas junções de tabelas. A utilização de *pipelining* distingue esta solução das anteriores, onde são aplicados aos planos de duas formas. A primeira consistia no seu uso para partilha de leituras das mesmas tabelas, aumentando a utilização de *cache*, e a segunda para a partilha de resultados intermédios entre planos. As heurísticas foram introduzidas para estimar e explorar estas propriedades comuns.

2.2.2.3 Abordagens por Agrupamento e Escalonamento

Como foi descrito anteriormente, as aplicações que precisam de efectuar diversos conjuntos de *queries*, podem beneficiar de ganhos significativos se estas puderem ser executadas concorrentemente, partilhando recursos.

Trabalhos efectuados nesta área [26] permitiram quantificar os potenciais ganhos esperados por quem utiliza este tipo de abordagem. No exemplo seguinte foram estudados os casos onde as *queries* são simples pesquisas por um valor chave (k), retornando o valor do registo como resultado.

Num primeiro cenário foi considerado o caso em que os dados (N registos) são guardados sequencialmente e de forma ordenada, por chave, em disco. Assumindo que são realizados k acessos de forma aleatória sobre N registos, provou-se que o ganho é dado por:

$$1 - \frac{2}{k+1} - \frac{2}{k(k+1)(N+1)}$$

O que significa que se estivermos a lidar com um volume de dados grande, o valor vai-se aproximar de:

$$1 - \frac{2}{k+1}$$

Para analisarmos melhor estes resultados, vejamos a seguinte tabela, que mostra o número médio de acessos ganhos e a conseguinte percentagem.

Batch size	Number of records in sequential file					
	100		1000		10000	
2	33.3	33.0	333.3	33.3	3333.3	33.3
5	168.1	66.6	1668.2	66.6	16668.2	66.6
10	413.1	81.8	4094.9	81.8	40913.1	81.8
20	913.8	90.5	9056.6	90.5	90485.2	90.5
50	2426.0	96.1	24043.6	96.1	240220.1	96.1
100	4950.0	98.0	49658.9	98.0	490148.0	98.0

Tabela 2.1: Número médio de acessos poupados com agrupamento de *queries* num ficheiro sequencial e a respectiva percentagem de ganho

Uma observação dos resultados mostra que, por exemplo, um agrupamento de 5 *queries* provocaria um ganho de 66.6%. Isto porque cada uma das 5 *queries* precisaria de consultar cerca de $1/2$ do ficheiro (em média, para encontrar um valor aleatório num ficheiro é necessário percorrer metade dos dados), o que daria um total de $5/2$ de passagens no total. De acordo com a fórmula a partilha dos dados provocaria uma redução de acessos para apenas $5/6$ dos registos.

O segundo cenário assume que os dados estão guardados em árvores n-árias com l níveis de profundidade, onde cada nó contém j apontadores e $j-1$ chaves. Assume-se também que as pesquisas são feitas de forma aleatória e que o resultado da pesquisa dentro de um nó é imediata.

Batch size	Number of levels in 11-ary tree							
	2		3		4		5	
2	1.1	28.1	1.1	18.9	1.1	14.1	1.1	11.2
3	2.2	38.7	2.3	26.2	2.3	19.6	2.3	15.6
5	4.7	49.1	4.9	33.7	4.9	25.2	4.9	20.1
10	11.8	61.4	12.5	43.1	12.6	32.4	12.6	25.8
20	28.3	73.7	30.1	53.0	31.2	40.0	31.3	31.9
50	84.0	87.6	95.2	65.6	97.6	50.0	97.9	40.0

Tabela 2.2: Número médio de acessos poupados com agrupamento de *queries* numa árvore 11-ária e a respectiva percentagem de ganho.

Como era esperado, a disposição dos dados em forma de árvore vai diminuir o ganho na aplicação mas, mesmo assim, estes valores continuam a ser bastante significativos. Este segundo cenário adapta-se bastante mais à realidade das bases de dados actuais, onde os dados são normalmente guardados em árvores B.

Como foi visto, o ganho teórico cresce com o número de *queries* que conseguimos agrupar e com o número de acessos às mesmas páginas. Contudo, este resultado está altamente dependente da memória disponível e dos tempos de resposta do sistema.

Os resultados foram demonstrados para um cenário muito pouco vantajoso para a partilha de recursos, onde apesar das pesquisas serem sobre a mesma tabela, os valores procurados são completamente independentes e aleatórios.

2.2.2.4 Mecanismos de *Cache* dos SGBD

Os SGBD, tal como os sistemas de operação, possuem um mecanismo de gestão de memória que permite guardar em RAM temporariamente alguns dos dados que foram acedidos mais recentemente, para que possam ser usados no futuro. Como o acesso a memória secundária (disco) é bastante mais lento, as operações sobre a base de dados serão mais rápidas, quanto menos vezes for necessário copiar essa informação para memória. Apesar de todo o processo ser automático, é interessante explorar as suas potencialidades, sobretudo se tivermos alguma flexibilidade para alterar a forma como os dados são enviados para processamento.

Normalmente os SGBD vão trazendo para memória, todos os dados que vão ser utilizados nas *queries* até esgotar todo o espaço disponível. Quando a memória chega ao limite torna-se necessário proceder à substituição de alguns desses blocos. Normalmente substituem-se os que não são referenciados há mais tempo, utilizando o algoritmo LRU (*Least Recently Used*), mas em certos casos isso não é verificado. O *Oracle* citeoracle, por exemplo, no caso dos *full table scans* não utiliza este método, pois torna-se improvável que os dados sejam utilizados novamente numa próxima operação (são colocados no final da lista LRU). No entanto, é possível alterar este comportamento alterando um atributo (CACHE/NOCACHE) nas tabelas.

2.2.2.5 Um *Middleware* para MQO

A solução apresentada em [27] consiste num *middleware* para explorar a presença de acessos partilhados de *queries* a tabelas ou índices de bases de dados. Como as optimizações são efectuadas fora do SGBD, este método torna-se independente da plataforma, podendo ser utilizado em diversas aplicações.

A ideia consiste em identificar *queries* com os mesmos interesses em termos de padrões de acessos (*teams*), juntando-as, e submetendo-as em simultâneo para a base de dados para serem executadas concorrentemente. Esta abordagem visa tirar partido dos mecanismos de *cache* dos servidores, por isso a sua eficácia será maior quanto maior for o conjunto de dados partilhados entre *queries*. Para se conseguir fazer o agrupamento é fundamental conhecer os detalhes dos acessos a memória secundária, particularmente nas *queries* e nos mecanismos de *cache*.

O *middleware* desenvolvido pode ser visto como um elo de ligação entre um canal onde são submetidas *queries* e os servidores da base de dados. A sua arquitectura é muito simples e consiste nos seguintes componentes: uma fila de *queries*, um módulo para escalonamento e selecção das *queries* na fila, um codificador de SQL e um módulo receptor de resultados.

É no entanto, no algoritmo de escalonamento, onde quase todo o processo relevante é efectuado. O algoritmo começa por seleccionar a *query* que se encontra no topo da fila (é apelidada de líder), de seguida vai percorrer as restantes posições e enquanto houverem processos suficientes na BD e *queries* com potencial para formar grupo com a líder, vai juntando elementos para a equipa. Retirando sempre a primeira *query* da fila garante-se que todas acabam por ser servidas eventualmente (*Liveness*).

Outro factor importante é o algoritmo que verifica o grau de afinidade entre *queries*. Duas *queries* são da mesma equipa se o algoritmo verificar se dado um certo limite (*threshold*), conseguem partilhar alguns interesses (p.e. mesmos acessos a dados), de acordo com uma regra de afinidade definida. Isto pode ser feito de várias formas, e talvez a forma mais simples, seja pensar que todas as *queries* que partilhem o mesmo template, sejam boas candidatas para formar equipa. Outras variantes podem envolver análise de comportamento, baseada em informação estatística recolhida empiricamente. Por exemplo, podemos imaginar que as *queries* que encaixem num determinado *template A*, provocam um ganho *X* na aplicação quando realizadas concorrentemente com as do *template B*.

2.3 Considerações Finais

Neste capítulo foram explicadas diferentes técnicas para optimização de compiladores. A sua utilização em linguagens com consultas SQL embutidas, pode não ser tão directa quanto nas linguagens habituais, mas é decerto um desafio interessante.

Na secção 2.1.1.2 foram apresentadas técnicas gerais para a propagação de informação em DFG. Estes métodos vão ser utilizados para a propagação de atributos alvo de afectações, até às respectivas operações de actualização de entidades. A definição das regras e do algoritmo vai seguir fielmente estas metodologias.

O compilador *OutSystems* implementa ainda os algoritmos de análise de variáveis activas (2.1.1.5) e de ordenação topológica (2.1.1.4) que foram apresentados neste capítulo. A existência destes algoritmos facilita o trabalho a realizar nestas optimizações, na medida em que minimiza o tempo total de implementação.

Todas estas optimizações e as optimizações em tempo de compilação no geral, acabam por ser um pouco limitativas. O conjunto de informação recolhido em tempo de compilação é muitas vezes de reduzido ou demasiado conservador. A combinação destas optimizações com informação recolhida em tempo de execução, pode trazer ganhos muito significativos para as aplicações e será certamente um dos grandes temas de investigação na área da compilação nos próximos anos [28].

No contexto desta dissertação a segunda parte deste capítulo não será tão relevante, porque a visão sobre os optimizadores de *queries* dos SGBD vai ser completamente transparente. Assim, se diferentes *queries* forem submetidas num único comando, vão ter planos de execução independentes. Isto deve-se ao facto dos SGBD utilizados na ferramenta (*SQL Server* [29] e *Oracle* [30]) ainda não suportarem optimizações ao nível múltiplas *queries*. Por outro lado, como a conexão por *action flow* é única, as *queries* mesmo quando submetidas em simultâneo, vão ser executadas em série, não tirando partido de uma eventual arquitectura de múltiplos processadores (a não ser quando uma única *query* utiliza mais do que um processador).

Desta forma, o envio simultâneo de *queries* apenas irá beneficiar do uso de *cache* e da redução na latência das conexões, fruto da redução do número de pedidos. No entanto, a evolução dos SGBD caminha no sentido de suportar a optimização de múltiplas *queries*, o que deixa boas perspectivas a que o trabalho apresentado nesta dissertação, seja futuramente recompensado sem ser necessário proceder a grandes ou mesmo nenhumas alterações.

Grande parte do trabalho desenvolvido em MQO, consiste em encontrar planos de execução que sejam mais eficientes do que a união de todos os planos óptimos individuais. As soluções que foram apresentadas mostraram algumas técnicas para contornar a complexidade do problema, utilizando pesquisas heurísticas e programação dinâmica. Contudo, quase todas implicavam criar camadas novas ou extensões aos optimizadores para definição de novos planos de execução.

Foram também identificadas algumas formas alternativas para contornar este problema. Estes meios tencionam explorar o uso de *cache* nos servidores, definindo regras para escalonamento das *queries* submetidas. Algumas dessas ideias foram mostradas em 2.2.2.5 e podem ser absorvidas neste trabalho. Dado um conjunto de *queries*, podemos ordenar a sua execução consoante regras de afinidade, que podem ir desde a simples partilha de tabelas, ou variações um pouco mais sofisticadas, como a verificação de correspondência de padrões. Pretende-se com isto explorar ao máximo o principio de localidade [11].

Para o ganho com agrupamento de *queries* ser maximizado, os grupos de *queries* precisam de ser aumentados. Uma adaptação do algoritmo de detecção de expressões activas permitirá identificar zonas no código para onde podem ser movimentados grupos de *queries*, sem introduzir mudanças semânticas nos programas.

A secção seguinte apresenta a plataforma *OutSystems*, os diferentes componentes e os aspectos mais importantes da linguagem. Depois são apresentados os padrões de código subóptimizados encontrados nas aplicações e descrito o devido processo de optimização. Neste processo foi dada especial atenção aquele que foi implementado na plataforma *OutSystems*.

3

Contexto e Problemas de Optimização

Neste capítulo é abordado o enquadramento do trabalho e as diferentes optimizações estudadas para a linguagem *OutSystems*. A secção inicial foca o contexto do problema e os diferentes componentes da *Agile Platform*, dando especial atenção à linguagem embutida no *Service Studio* e às suas construções mais relevantes. Na secção seguinte é mostrada a problemática da coexistência entre consultas e comandos SQL e lógica aplicacional numa linguagem de programação, e as diferentes situações que criam padrões sub-optimizados. De seguida, são apresentados alguns padrões encontrados nas aplicações e explicadas as optimizações propostas. Por fim, é descrito o padrão que foi escolhido para ser optimizado no compilador *OutSystems*.

3.1 Enquadramento

Esta dissertação insere-se no contexto de uma colaboração entre o grupo investigação de linguagens de programação do CITI (Centro de Informática e Tecnologias de Informação) do Departamento de Informática da FCT/UNL e a empresa *OutSystems*.

A *OutSystems* [1] é uma empresa de *software* que disponibiliza uma ferramenta para produção de aplicações *Web* que evoluem com o tempo. A *Agile Platform* apresenta meios para a criação e manutenção de aplicações *Web* de uma forma rápida e incremental. É possível obter soluções funcionais num curto espaço de tempo e ir acrescentando novas funcionalidades à medida que as necessidades aparecem. Torna-se assim fácil apresentar versões provisórias, e modificá-las de acordo com o *feedback* dos clientes ao longo do tempo, aumentando o ciclo de vida dos programas realizados na ferramenta.

O principal objectivo da empresa consiste em disponibilizar um meio de desenvolvimento

que permita que os programadores se abstraiam de aspectos sintácticos e de integração de diversos componentes (com diferentes interfaces), focando a sua atenção na lógica das aplicações.

3.2 *Agile Platform*

A *Agile Platform* é uma ferramenta *All-in-One* que permite a rápida criação de aplicações *Web* empresariais construídas para estar em constante mudança. A plataforma disponibiliza meios de suporte ao ciclos de vida destas aplicações permitindo a rápida criação, administração e também integração de componentes. O grande objectivo desta abordagem é acelerar a entrada dos produtos no mercado e acompanhar as suas necessidades com flexibilidade, sem comprometer o controlo.

A plataforma está dividida em quatro componentes:

Service Studio é uma aplicação que permite a criação de aplicações *Web* de raiz, que podem integrar componentes externos, utilizando modelos visuais. A linguagem interna do *Service Studio* é o principal foco deste trabalho.

Integration Studio permite aos utilizadores criar componentes próprios e integrar em aplicações externas estendendo as funcionalidades e o modelo de dados. Esta integração é cumprida através de extensões: um conjunto de acções, entidades e estruturas disponíveis em *Service Studio* mas implementadas em tecnogias externas.

Service Center é uma consola Web que permite a administração completa da Agile Platform. Possibilita, entre outras coisas, a definição de políticas de acesso entre as equipas de desenvolvimento.

Embedded Change Technology é um mecanismo para recolher o feedback dos utilizadores directamente das aplicações desenvolvidas.

A figura 3.1 ilustra a disposição destes quatro componentes na arquitectura do desenvolvimento.

3.2.1 *Service Studio*

O *Service Studio* (mostrado na figura 3.2) oferece ao programadores de aplicações *Web* tanto a capacidade de modelar visualmente o aspecto das páginas, como a lógica aplicacional ligada à apresentação dos objectos e às acções dos utilizadores finais. Este conjunto de funcionalidades permite que idealmente os programadores não tenham de se preocupar em alterar ou escrever o código gerado pela plataforma.

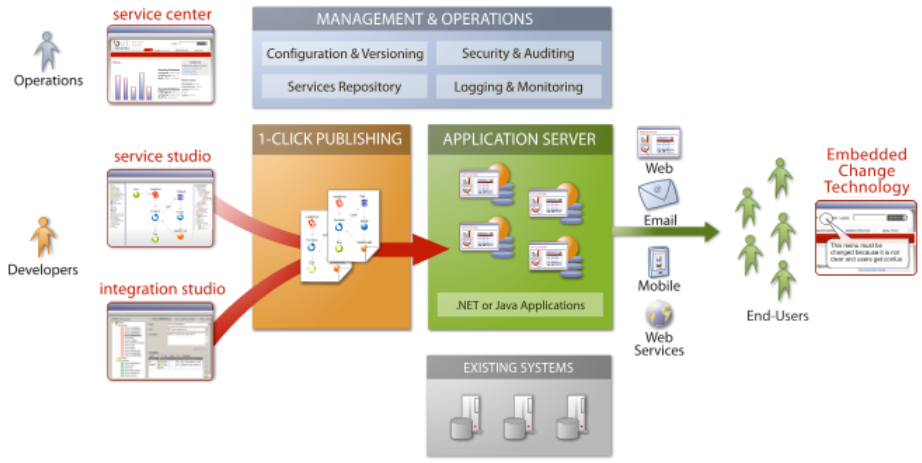

Figura 3.1: Arquitectura da plataforma *OutSystems*

A criação de aplicações tem como base ficheiros OML (*OutSystems Markup Language*), denominados vulgarmente por *eSpace*, que contêm toda a informação da aplicação desenvolvida com excepção do código das referências externas.

Estes ficheiros podem ser editados individualmente por cada programador e o trabalho conjunto posteriormente condensado num só ficheiro através de um mecanismo de junção (*merge*) interno ao *Service Studio*. Este permite ainda a verificação de consistência do modelo que procura sobretudo erros de tipificação e de sintaxe. Após a verificação pode-se iniciar o processo de compilação que transforma o código gerado no ficheiro OML em código C# ou Java (dependendo da versão), publicando-o no servidor.

Esta operação no entanto não é interna ao *Service Studio*. Quando o programador invoca o método de publicação (1CP: *one-click publish*), o *Service Studio* contacta o *Service Center*, que monitoriza a operação, onde são realizadas as operações de compilação e publicação da aplicação, ficando posteriormente disponível para uso.

O *Service Studio* permite definir:

- Interface do utilizador
- Lógica aplicacional
- Modelo de dados
- Temporizadores
- Fluxo de trabalho

Como produto final típico de uma aplicação desenvolvida no Service Studio resulta um conjunto de páginas Web, com tabelas que listam elementos de bases de dados, e operações para

manipular esses mesmos dados. Estes componentes têm os seus comportamentos associados que podem ser executados em concorrência pelos diversos utilizadores da aplicação final.

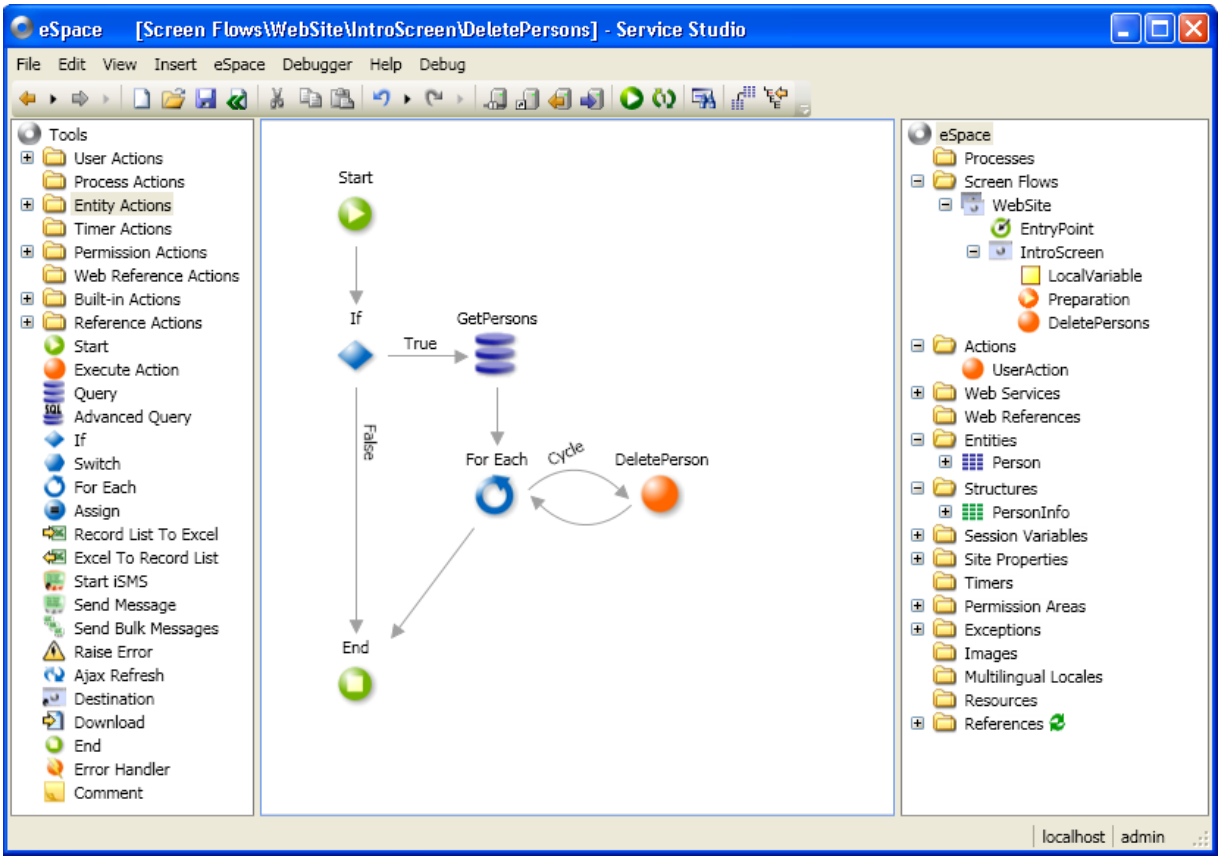

Figura 3.2: Aparência do *Service Studio*

3.3 A Linguagem

O *Service Studio* implementa uma linguagem de domínio específico (DSL) que permite especificar o comportamento dos componentes das aplicações desenvolvidas. As interacções dos utilizadores com os elementos das páginas desencadeiam procedimentos para tratamento desses eventos (*Actions*), cujos comportamentos são especificados através da modelação de *action flows*. Os *action flows* representam um conjunto de operações ligadas sobre a forma de um grafo orientado, potencialmente cíclico, que contém um nó inicial e um conjunto de nós terminais. Estas acções podem ter um conjunto de parâmetros de entrada, saída e podem definir varáveis locais auxiliares.

Os restantes nós do grafo representam diversas operações, desde as habituais escolhas e atribuições até a nós especiais para *queries*, iteração de listas ou navegação entre páginas. Os programadores, podem ainda criar as suas próprias acções (*User Actions*) ou referenciar acções de *eSpaces* (módulos) externos.

A definição dos ecrãs é feita através de um editor gráfico que permite arrastar componentes para a página em construção. Aqui, o aspecto mais importante, é que estes ecrãs são vistos pelo compilador como acções que podem também ter parâmetros de entrada e de saída, assim como variáveis locais.

3.3.1 Construções da Linguagem e *Action Flows*

Os *Action Flows* permitem a modelação de acções através da união sequencial de operações. Essas operações vão deste um conjunto restrito de comandos e acções predefinido, até à possibilidade de importar acções externas à aplicação. Pode-se acrescentar a esta lista o conjunto de acções definido pelo programador. A execução de uma única acção pode desencadear assim a passagem por vários *action flows*.

De seguida são descritas de forma sumária as principais construções da linguagem integrada no *Service Studio*. A listagem de nós é a seguinte:

Start & End - Delimitam um *action flow*.

Assign - Permite fazer atribuição de valores a um conjunto de variáveis.

If & Switch - Controlam o fluxo do programa através da avaliação de expressões.

Simple Query - Executa uma consulta à base de dados e retorna uma lista de valores de uma entidade ou união de entidades. A criação da *query* é auxiliada por uma interface gráfica (Figura: 3.4), que após a introdução dos dados (parâmetros, entidades, condição de junção e de ordenação), cria a consulta de forma automática.

Advanced Query - Esta operação é semelhante à anterior mas aqui cabe ao utilizador a responsabilidade de criar o código da consulta numa linguagem próxima do *standard SQL*.

Foreach - Este nó itera uma lista e cria um ciclo de acções que são executadas para cada elemento dessa iteração. Este nó tem sempre uma ligação de saída e de entrada suplementares porque, para além habituais, engloba um fluxo de ciclo.

Destination - Devia o fluxo de execução de um *action flow* para uma página *Web*.

Error Handler - Captura uma excepção e executa um fluxo de tratamento.

Excel to/from RecordList - Recolhe ou envia dados para uma folha de cálculo Excel. Utiliza tipos *RecordList* para guardar ou enviar essa informação.

Execute Action - Nó que executa uma determinada acção (Figura 3.4). Entre estas destacam-se: *Entity Actions*, *User Actions*, *Referenced Actions* e *Built-In Actions*.

Entity Actions - Conjunto de acções criadas no momento da definição de uma entidade. As acções criadas são as seguintes: *Create<Entity>*, *CreateOrUpdate<Entity>*, *Update<Entity>*, Delete<Entity>, *Get<Entity>* e *GetForUpdate<Entity>*. Esta última, bloqueia a entidade na base de dados até que os valores sejam actualizados por um *Update<Entity>*. A acção *CreateOrUpdate<Entity>* procura primeiro actualizar a entidade e caso esta não exista, cria uma nova.

User Actions - Conjunto das acções definidas pelo utilizador que podem ser reutilizadas no programa.

Referenced Actions - Acções externas ao *eSpace* importadas para uso local.

Built-In Actions - Conjunto de acções de sistema disponíveis para os programadores.

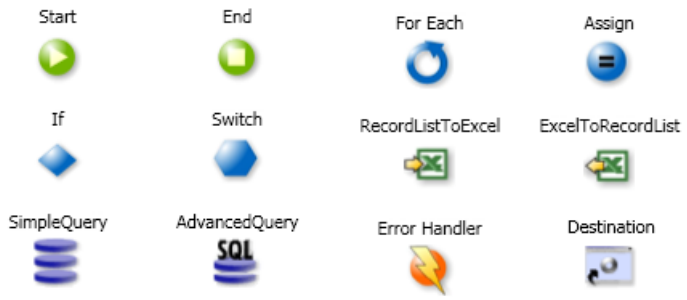

Figura 3.3: Principais nós da linguagem *OutSystems*

A figura 3.6 mostra um exemplo dum *action flow* que calcula o resultado da função factorial para o valor 10. A execução parte do nó de Start em direcção a dois nós de afectação para inicialização dos valores das variáveis. O nó de decisão vai originar um ciclo devido às ligações dos nós seguintes e a execução do action flow termina quando se atinge o nó End.

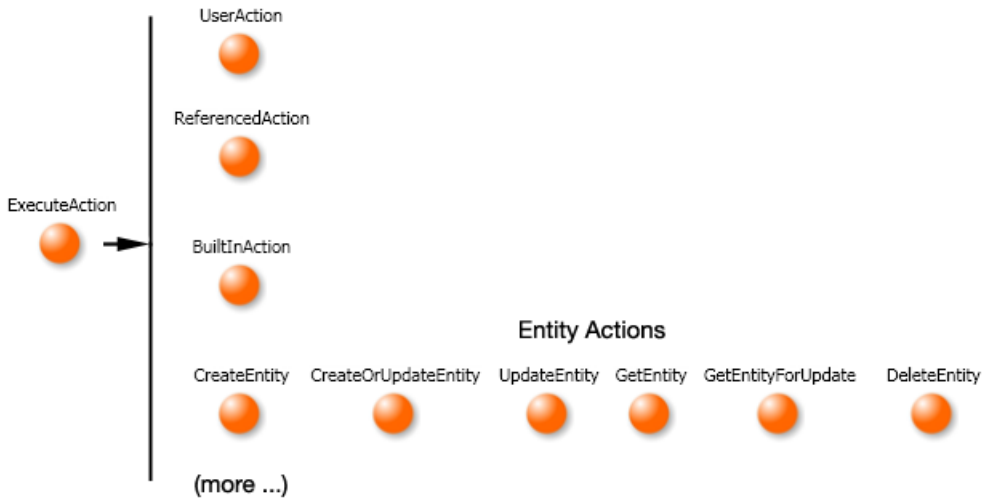

Figura 3.4: Exemplos dos nós *Execute Action* mais relevantes

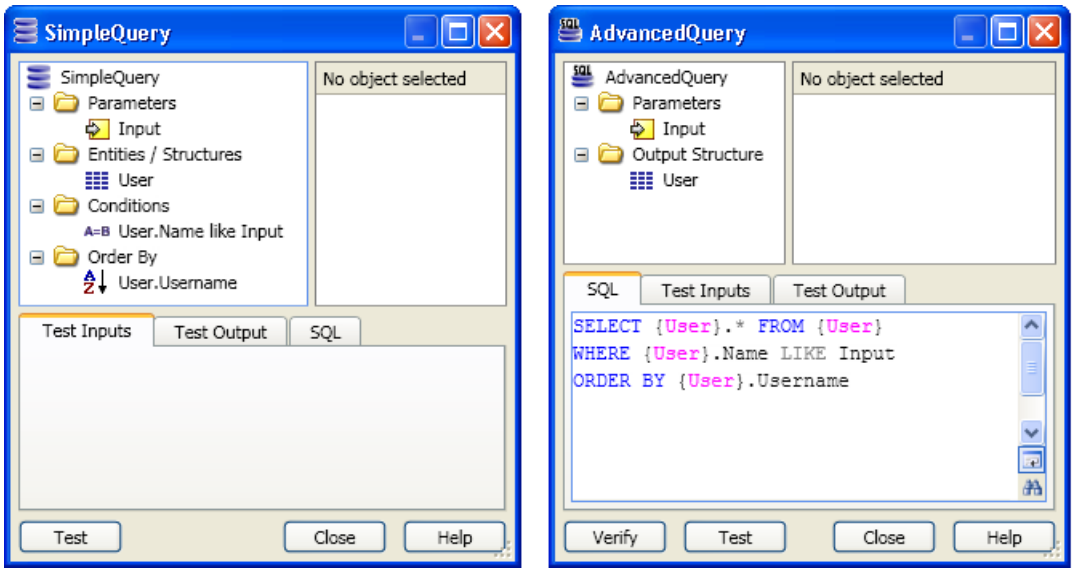

Figura 3.5: Editores de *Queries* (*Simple* e *Advanced* respectivamente)

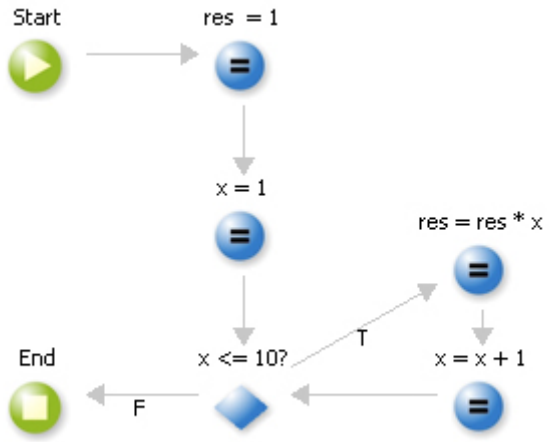

Figura 3.6: Exemplo dum *action flow* para a função factorial

3.3.2 Interacção com as Bases de Dados

Toda a interacção com as bases de dados é efectuada através de construções próprias da linguagem, com parametrização fortemente tipificada, permitindo detecção prévia de erros de execução. A ferramenta permite a comunicação com bases de dados definidas em *Microsoft SQL Server* [29] e *Oracle* [30], utilizando respectivamente os níveis de isolamento *Read Uncommited* [15] e *Read Commited* [15]. Em ambos os casos este atributo não pode ser configurável.

No final de cada action flow, as alterações sobre a base de dados são automaticamente alvo de um *commit*. No entanto, existe a possibilidade de, através de uma outra operação da linguagem, fazer commit explicitamente a qualquer altura sobre os dados alterados. Da mesma forma, se algo correr mal é possível forçar que seja efectuada uma operação de *rollback* trazendo o sistema de volta ao estado anterior.

3.3.3 Tipos de Dados

A linguagem *OutSystems* permite a criação e manipulação de um grande conjunto de tipos de dados. Para além dos tipos básicos de qualquer linguagem de programação (*integer*, *decimal*, *boolean*, *string* e *byte[]*) existe um conjunto dados próprios da linguagem (p.e. *DateTime*, *Date*, *Time*, *PhoneNumber* e *Currency*).

O *Service Studio* permite também a definição de entidades. Estas são usadas para definir o modelo de uma base de dados relacional associada à aplicação. As entidades podem ser constituídas apenas por um identificador e por qualquer um dos tipos (não compostos) indicados anteriormente. Cada instância de uma entidade é representada por uma instância do tipo *EntityRecord*.

Para a estruturar os dados em memória existe a possibilidade de definir estruturas para agrupar conjuntos de dados relacionados. As estruturas permitem maior flexibilidade que as entidades nos tipos de dados dos atributos, podendo definir atributos de qualquer tipo, sendo a única restrição a impossibilidade de definição de estruturas recursivas (com atributos do próprio tipo).

Para colmatar a necessidade de composição de dados, existe ainda a oportunidade destes serem agrupados em registos (*Records*) e em listas de registos (*RecordLists*). Os registos permitem a criação de conjuntos de entidades e estruturas, sendo também usados para definir o tipo de cada elemento dos resultados duma consulta à base de dados. Uma *query* que resulte na junção de *N* tabelas, retorna uma *RecordList* de tipos *Record* compostos por *N* elementos do tipo *EntityRecord*.

Cada *RecordList* é definida internamente por um vector de registos, tendo a capacidade de indexação, e possuindo ainda a noção de apontador para o registo corrente (*Current Record*).

3.4 Optimizações

Os níveis de abstracção oferecidos aos programadores de uma DSL normalmente reflectem-se em código mal estruturado e redundâncias no resultado final da compilação. Nas linguagens mais avançadas, existe um optimizador de código responsável por analisar a estrutura do programa e introduzir as optimizações necessárias para resolver este tipo de situações. Neste caso em particular o código é optimizado, no entanto, ainda é possível melhorar a performance do sistema em alguns aspectos.

Foi identificado um conjunto de operações que pode ser enviado para os SGBD em simultâneo, reduzindo substancialmente o número de ligações aos servidores e retirando algum do *overhead* introduzido pelo número actual de pedidos. O envio em grupo de *queries* pode ter um impacto duplamente proveitoso para o sistema, uma vez que algumas das destas podem partilhar tabelas entre si. Se uma *query* começar e algumas das tabelas da *query* anterior ainda estiverem em memória, a segunda pode beneficiar do sistema de *cache* do servidor 2.2.2.4. Para tal, torna-se necessário resolver os problemas de dependências entre as operações, utilizando análise de fluxo de dados, para verificar quais podem ser enviadas em grupo, e finalmente para tirar melhor partido da cache, encontrar uma ordem para a execução dentro do grupo.

Os níveis de comunicação entre as aplicações e os sistemas de gestão de bases de dados afectam bastante a performance e acrescem consideravelmente se as bases de dados não forem locais à aplicação. Pensa-se assim que todas as optimizações que minimizem os custos de comunicação são interessantes de explorar. No entanto este tipo de optimizações, que por vezes implica antecipar consultas ou até mesmo eliminar algumas, podem facilmente trazer problemas de inconsistências de dados. A solução muitas vezes consiste em chegar a um compromisso entre consistência e grau de comunicação.

Algumas das práticas comuns entre programadores podem também estar na origem do aparecimento de padrões sub-optimizados e estão geralmente ligadas à facilidade de compreensão do código ou à falta de expressividade da linguagem. Assim, surgem outras optimizações que

têm como base a construção de *queries* mais compactas, eliminando algumas com operações equivalentes, e o reconhecimento de operações redundantes. As *queries* podem aparecer repetidas ao longo de um fluxo de execução ou até mesmo dentro de ciclos. Estas últimas representam um aspecto importante, porque derivado da concorrência entre operações, podem produzir resultados diferentes ao longo do tempo.

O código sub-optimizado pode também resultar da existência de código genérico proveniente do elevado nível de abstracção. As operações sobre entidades geralmente manipulam-nas como um todo, mesmo quando são apenas necessários alguns dos seus atributos. A redução, em tempo de compilação, do número de atributos usados em operações sobre entidades, pode ter um impacto muito significativo na comunicação com o SGBD. No entanto, para o compilador conseguir filtrar os atributos necessários em cada uma destas operações, necessita de propagar informação sobre os dados que serão futuramente úteis, utilizando técnicas de análise de fluxo de dados 2.1.1.2.

Na subsecção seguinte são enumerados os padrões encontrados em algumas aplicações e é descrito o respectivo processo de re-estruturação.

Os exemplos mostrados nas subsecções seguintes encontram-se em *SQL Server* e C#.

3.4.1 Cenários de Optimização

Nesta subsecção são apresentados os diferentes cenários de optimização encontrados na fase inicial deste projecto. Nenhum destes cenários foi escolhido para ser implementado na plataforma (devido a limitações de tempo), não obstante a secção esta secção apresenta todos estes padrões e expõe brevemente a devida optimização proposta.

3.4.1.1 Agrupamento de *Queries* Independentes

Quando, num determinado ponto de um *action flow*, existem *queries* independentes e que estão ligadas entre si pelos arcos do grafo, podemos agrupá-las num único pedido sobre a base de dados. Duas *queries* são designadas independentes quando em nenhuma das duas há parâmetros de entrada provenientes dos resultados da outra, ou produto do processamento desses mesmos dados.

Actualmente, para cada *query* encontrada no *action flow*, é gerada uma função de código C# com a sua construção e colecção dos respectivos resultados obtidos (que são posteriormente retornados num objecto do tipo *RecordList*). O objectivo deste agrupamento é construir apenas uma função, onde o código de todas as *queries* é enviado num único comando SQL, e onde os resultados são coleccionados numa lista de objectos do tipo *RecordList*, que será o retorno da função.

A figura 3.7 actualmente geraria código SQL e C# para duas consultas diferentes e dois iteradores independentes, como mostra de forma simplificada a figura seguinte:

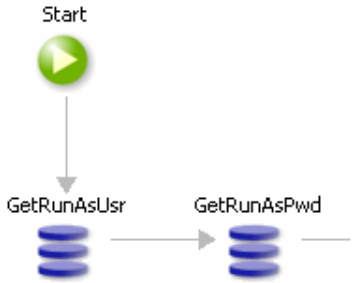

Figura 3.7: Exemplo de agrupamento de *queries* independentes

```
// GetRunAsUsr sample source
SqlCommand command = new SqlCommand("SELECT Id, Espace_Id, Name, Val FROM Espace_Configuration
                                     WHERE Espace_Id = " + Id + " and ToLower(Name) = "+ Name1, connection);
using (SqlDataReader reader = command.ExecuteReader()) {
    while (reader.Read()) {
        Id1 = (int)reader.GetValue(0);
        /* .. .*/
    }
}

// GetRunAsPwd sample source
SqlCommand command2 = new SqlCommand("SELECT Id, Espace_Id, Name, Val FROM Espace_Configuration
                                     WHERE Espace_Id = " + Id + " and ToLower(Name) = " + Name2, connection);
using (SqlDataReader reader2 = command2.ExecuteReader()) {
    while (reader2.Read()) {
        Id2 = (int)reader2.GetValue(0);
        /* .. .*/
    }
}
```

A optimização proposta tem como objectivo unir ambas as consultas numa única instrução SQL separando-as simplesmente com o sinal ";". O código seguinte mostra como poderiam ser obtidos os resultados de duas *queries* separadas desta forma.

```
// Optimized sample source
SqlCommand command = new SqlCommand("SELECT Id, Espace_Id, Name, Val FROM Espace_Configuration
                                     WHERE Espace_Id = " + Id + " and ToLower(Name) = "+ Name1 +
                                     ";SELECT Id, Espace_Id, Name, Val FROM Espace_Configuration
                                     WHERE Espace_Id = " + Id + " and ToLower(Name) = " + Name2, connection);
using (SqlDataReader reader = command.ExecuteReader()) {
      while (reader.HasRows) {
          while (reader.Read()) {
               Id1 = (int)reader.GetValue(0);
               /* .. .*/
          }
          reader.NextResult();
      }
}
```

Na realidade a geração de código de leitura não está centralizada. Cada umas destas consultas idealmente ficaria com o seu *reader* associado na função de leitura da *RecordList* para que os dados pudessem ser iterados independentemente. O problema desta solução é que o *DataReader* funciona como um iterador sequencial dos resultados de uma consulta SQL (múltiplas neste caso) e instruções de iteração aninhadas, envolvendo as duas *queries*, iriam provocar problemas nas iterações. A solução passará por recolher dados o mais cedo possível iterando as listas com informação local. Esta implicação teria que ser avaliada e posteriormente encontrados os cenários onde seria vantajoso aplicar esta técnica.

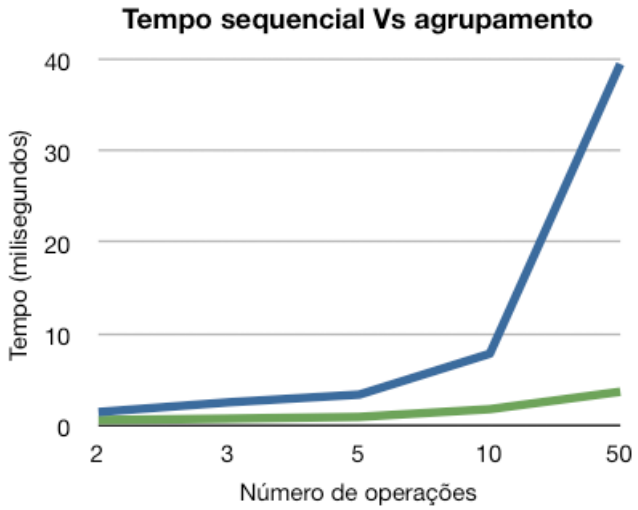

Figura 3.8: Diferença temporal em milisegundos entre envio sequencial (azul) e em simultâneo (verde)

Alguns testes realizados mostraram que o peso das ligações pode ser um factor importante a

explorar neste trabalho. Os resultados da execução de múltiplas consultas semelhantes, efectuadas num servidor de rede local onde a latência nas mensagens é muito reduzida, mostrou que quando os resultados estão em *cache* e são de pequena dimensão, o factor com maior peso é a ligação com o servidor.

A figura 3.8 mostra que que para as consultas sequenciais, os valores aumentam num factor próximo de 1:1 enquanto que no segundo os valores são muito mais baixos (próximos de 1:4), embora em ambos os caso os valores sejam bastante reduzidos.

Aparentemente os resultados são pouco significativos (poucos milisegundos), mas se os dados se encontrarem em bases de dados cuja latência de comunicação é muito superior, os efeitos vão ser mais notados, principalmente se os dados estiverem em cache (no servidor) ou se os resultados forem de pequena dimensão comparativamente à execução do pedido.

Para se detectar quando é que um conjunto de *queries* pode ser enviado para a base de dados em simultâneo é necessário verificar as suas dependências internas. O algoritmo apresentado em 2.1.1.6, permite verificar as dependências entre as *queries* e encontrar zonas de código onde essas expressões podem ser avaliadas em conjunto, mantendo a mesma semântica e conjunto de resultados. Torna-se necessário proceder a algumas adaptações porque não são apenas as afectações que alteram os potenciais resultados das *queries*. Algumas *Entity Actions*, criadas no momento da geração da entidade, e as *Advanced Queries* podem alterar esses dados e devem modificar a lista *kill* (2.1.1.2). O mesmo acontece com os nós *Execute Action* que actualmente escondem o que acontece no seu interior, podendo conter qualquer uma das operações anteriores.

Depois de reunidos estes conjuntos de *queries* existem várias hipóteses possíveis de execução, uma vez que cada *query* poderá pertencer a vários grupos (cada nó terá um conjunto de *queries* que pode ser executado nesse ponto). A selecção dos diferentes conjuntos não é trivial, pois o simples cálculo dos grupos de maior dimensão é um problema desafiante. Em particular se tivermos em conta que algumas *queries* podem partilhar tabelas e ter melhor desempenho quando executadas em simultâneo.

A solução poderá passar por um algoritmo de programação dinâmica, para gerar toda a combinatória sem recalcular conjuntos, que considere heurísticas de afinidade entre *queries* (2.2.2.2). Contudo, o custo de execução poderá não compensar os ganhos relativamente baixos obtidos nos testes.

3.4.1.2 União de Duas *Queries* Dependentes com *Left Join*

Nem só as *queries* independentes podem ser agrupadas numa única operação sobre a base de dados. Quando existe uma *query* que retira um conjunto de valores que vai ser usado como parâmetro numa outra, é possível agrupar as duas operações numa única, utilizando um *left join*, que vai retornar o resultado das duas *queries* originais. Esta técnica pode ser vista como uma adaptação do conceito de propagação por cópia (2.1.3.2) à realidade das bases de dados. Na verdade não tem o mesmo significado, porque os resultados acabam por ser guardados numa

variável após a consulta seguinte. A grande vantagem é que se poupa um acesso à base de dados e, se os dados da primeira consulta não forem necessários mais à frente, podem ser optimizados e consequentemente descartados.

A detecção deste padrão poderá passar pela propagação dos identificadores definidos pelas *queries* para a frente no grafo. Se esses identificadores chegarem a uma outra *query* sem que nenhum deles tenha sido lido ou escrito no caminho, então podem ser recolhidos numa única operação. Na prática, isto significa que estes identificadores não podem estar e nenhuma lista *def* ou *use* dos nós do caminho, e caso algum deles esteja, as *queries* não podem ser colocadas sequencialmente e a transformação não será aplicada.

Para efeitos de simplificação, este padrão considera apenas as consultas que não são futuramente iteradas, para que os iteradores das *RecordLists* não tenham que ser alterados ou os resultados antecipados. É possível obter esta informação com a versão actual do compilador.

O padrão descrito pode ser identificado na imagem 3.9.

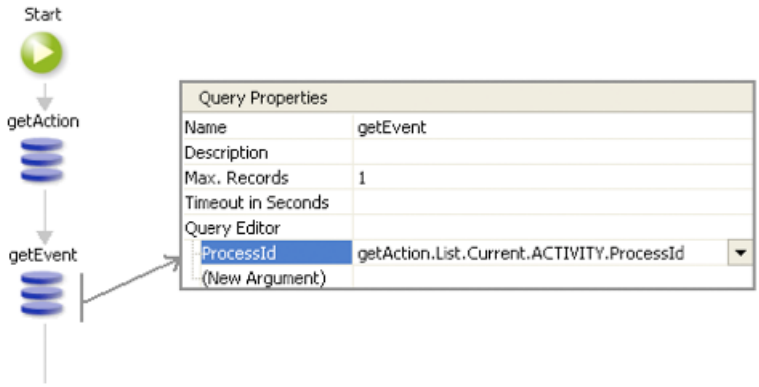

Figura 3.9: Exemplo do padrão a aplicar *left join*

Tal como no exemplo anterior, o código gerado actualmente consiste em duas funções diferentes e independentes, que executam as *queries* e compilam os resultados num objecto *RecordList*. O exemplo seguinte mostra, para o *action flow* da figura 3.9, uma simplificação do código gerado actualmente:

```
// getAction sample source
SqlCommand command = new SqlCommand("SELECT ACTIVITY.ProcessId FROM ACTIVITY
                                     WHERE ACTIVITY.Id = " + Id, connection);
using (SqlDataReader reader = command.ExecuteReader()) {
    while (reader.Read()) {
        processId = (int)reader.GetValue(0);
    }
}
// getEvent sample source
SqlCommand command2 = new SqlCommand("SELECT EVENT.name FROM EVENT
                                     WHERE EVENT.ProcessId = " + processId, connection);
using (SqlDataReader reader2 = command2.ExecuteReader()) {
    while (reader2.Read()) {
        eventName = (string)reader2.GetValue(0);
    }
}
```

Depois da devida transformação os dois valores podem ser obtidos numa única função da seguinte forma:

```
// Optimized sample source
SqlCommand command = new SqlCommand("SELECT ACTIVITY.ProcessId, EVENT.name FROM ACTIVITY
                                     LEFT JOIN EVENT ON ACTIVITY.Id = EVENT.ProcessId
                                     WHERE ACTIVITY.Id = " + Id, connection);
using (SqlDataReader reader = command.ExecuteReader()) {
    while (reader.Read()) {
        processId = (int)reader.GetValue(0);
        eventName = (string)reader.GetValue(1);
    }
}
```

A opção pela operação *left join*, resulta da possibilidade de não existirem valores na segunda *query* para os parâmetros retornados na primeira. A operação *left join* permite que no lado direito da junção sejam obtidos valores nulos mesmo que não haja correspondência, retornando sempre os valores da primeira *query*.

3.4.1.3 Agrupamento de Operações de Inserção em Ciclos

Uma das optimizações mais interessantes consiste no agrupamento de operações de inserção em ciclos, onde os novos elementos são unidos numa única instrução de inserção no final de um ciclo. O padrão (Figura 3.10) consiste na iteração de uma *RecordList* com a instrução *For Each*, onde a cada passo existe uma operação de inserção no ciclo desta instrução.

Esta abordagem trás, à semelhança das anteriores, algumas implicações semânticas: os dados passam a estar disponíveis mais tarde para os utilizadores e se for detectada alguma excepção, não será capturada na altura da inserção correspondente, o que poderá não ser o comportamento desejado.

No entanto, testes prévios (indicados mais à frente) mostraram que as perspectivas de ganho são bastante elevadas, o que faz com que esta alteração se torne importante nem que a sua invocação seja feita explicitamente pelos programadores.

A transformação só poderá ser aplicada se os valores inseridos não forem utilizados em outras operações dentro do mesmo ciclo. Da mesma forma, os identificadores gerados pela base de dados que são o retornados na acção, não podem ser usados no decorrer do ciclo. A plataforma poderá ainda sugerir ao programador, através de uma mensagem de aviso, que as operações podem ser unidas no final do ciclo para maior eficiência.

A abordagem sugerida, passa por invocar a operação explicitamente através da introdução de uma propriedade na operação *Create<Entity>* que permitirá definir o comportamento da inserção. Alternativamente, esta propriedade poderá ser definida no próprio ciclo agrupando todas as inserções existentes no seu interior.

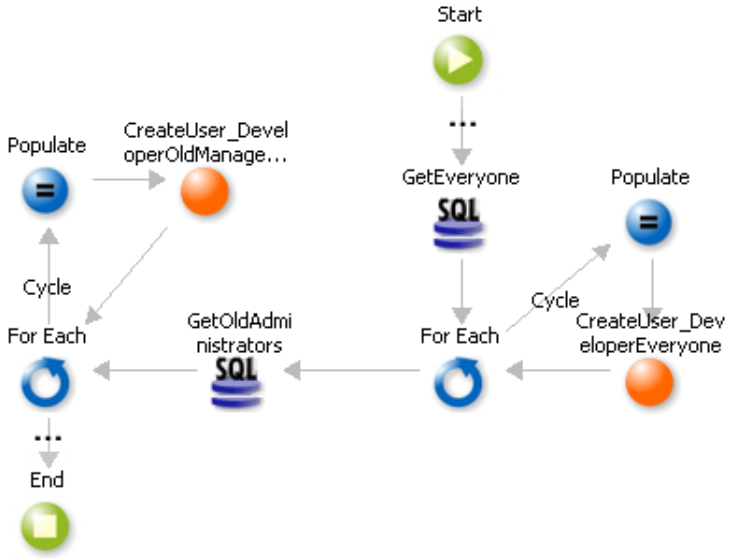

Figura 3.10: Exemplo do padrão de inserção em ciclos

A ferramenta já disponibiliza as primitivas necessárias (em C#) para agrupar as operações de inserção (*bulk inserts* [31]) tanto em *SQL Server* como em *Oracle*, que não são directamente suportados pelo SQL. Estas operações seriam invocadas imediatamente antes da saída da operação *For Each* no seu código gerado. Os nós *For Each* passariam então a suportar listas de inserções, que seriam alteradas pelas operações *Create<Entity>* correspondentes, e futuramente

unidas numa instrução e executadas à passagem por esse nó em tempo de execução.

Os testes realizados permitiram quantificar as diferenças temporais entre as inserções singulares sobre a base de dados em relação ao método que utiliza as funções de *bulk insert* do C# para *SQL Server*. No exemplo seguinte foram inseridos vários números aleatórios numa tabela (um por tuplo).

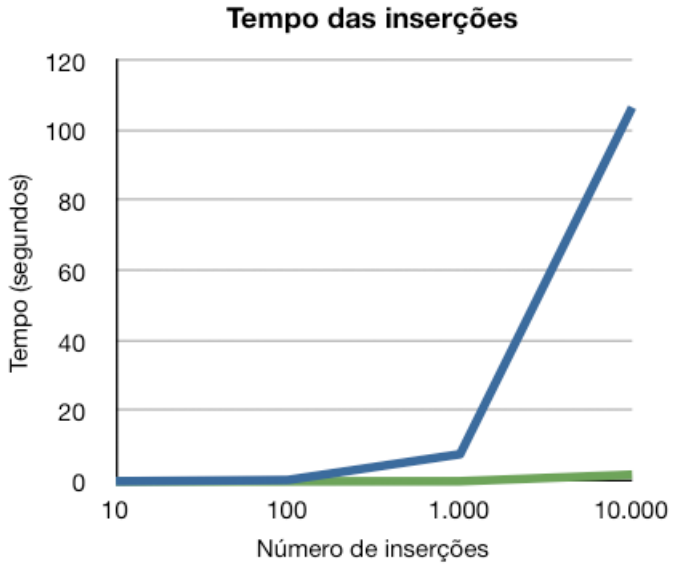

Figura 3.11: Diferença temporal em segundos entre inserções singulares (azul) e em simultâneo (verde)

Nº de inserções	10	100	1.000	10.000
Tempo Singulares (s)	0.05	0.44	7.78	106.76
Tempo Simultâneo (s)	0.02	0.03	0.06	1.94

Tabela 3.1: Diferença de tempo em segundos entre inserções em conjunto e singulares

Os resultados dos testes foram bastante conclusivos. Podemos observar que os ganhos começam a ser significativos mesmo com poucas inserções e vão aumentando a relevância com o aumento do volume de dados. Isto pode ser especialmente importante em aplicações onde o carregamento de dados é uma funcionalidade comum e onde geralmente a ordem de grandeza de valores é elevada. Portanto, não é de todo irrealista pensar que possam existir casos onde sejam criados mais de 1000 elementos na base de dados num único ciclo, e neste caso, os ganhos

podem passar as dezenas de segundos.

3.4.1.4 Antecipação de Resultados de *Queries*

Este é um dos casos mais interessantes de analisar e ao mesmo tempo um dos com melhores perspectivas de bons resultados. O padrão consiste numa consulta à base de dados, seguido de uma iteração desses mesmos resultados onde, para cada um deles, é feita uma nova consulta. Este modelo está normalmente associado a pesquisas por identificadores e uso dos mesmos para ir buscar valores a uma tabela diferente. Para efeitos de simplificação, apenas vamos considerar os casos onde só será utilizado o primeiro registo dos resultados na *query* interior, ou seja, quando o compilador detectar que não existe iteração ou acesso a outro índice que não o primeiro (verificação implementada actualmente). Os casos em que isto não acontece são muito menos frequentes e de complexidade bastante mais elevada.

A figura seguinte ilustra um exemplo do padrão encontrado:

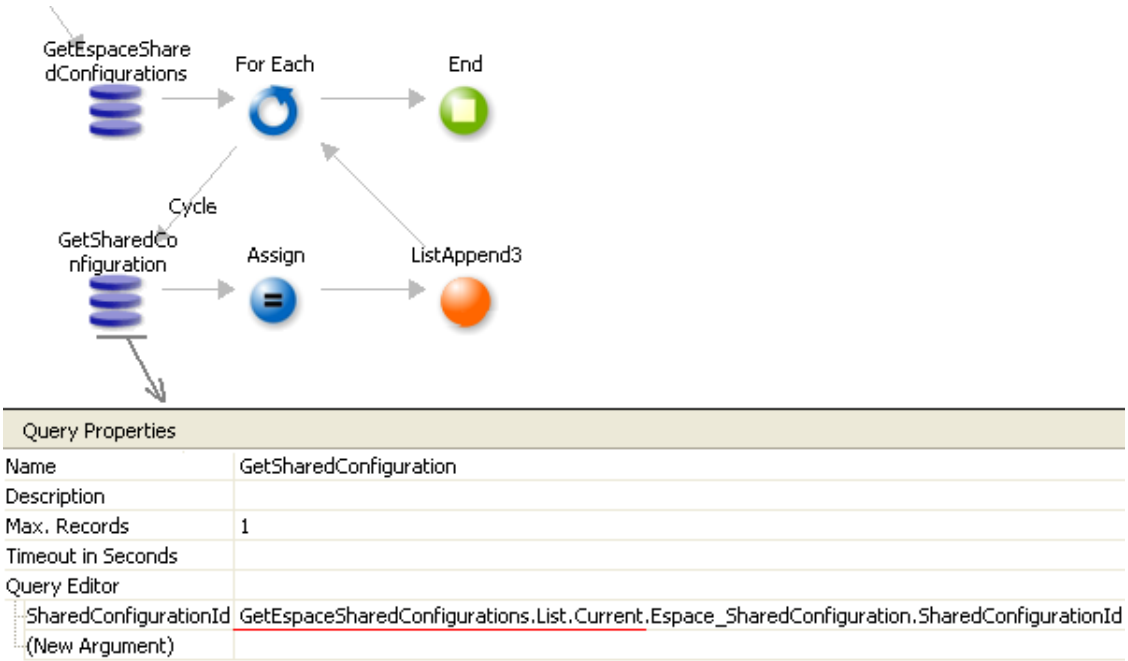

Figura 3.12: Exemplo do padrão para pré-cálculo de *queries*

Para situações onde os limites do ciclo *For Each* não são definidos, o número de consultas efectuadas dentro do ciclo vai ser igual ao número de resultados obtidos na consulta anterior. Este valor pode ser substancialmente reduzido se os resultados forem obtidos de uma única vez e depois efectuada uma busca pelo resultado com o identificador pretendido em memória local,

a cada passo da iteração.

Internamente, esta optimização pode ser vista como uma consulta anterior ao ciclo que vai buscar todos os resultados possíveis (Fig. 3.13 - A) conjugada com uma operação de pesquisa nesses mesmos resultados (Fig. 3.13 - B). As duas *queries* podem ser ainda juntas numa única operação através da operação *left join* para garantir que todos os resultados do lado esquerdo são capturados mesmo que não haja resultado na segunda *query* para esse valor, à semelhança do que foi proposto no padrão 3.4.1.2.

É fundamental também verificar se existem escritas no interior do ciclo sobre a tabela onde está a ser efectuada a leitura interior ao ciclo. O compilador também poderá efectuar esta operação a pedido do utilizador, informando-o da oportunidade de pre-calcular os resultados e de usar *cache*. Para este caso seria necessário acrescentar uma propriedade no ciclo para seleccionar o comportamento desejado.

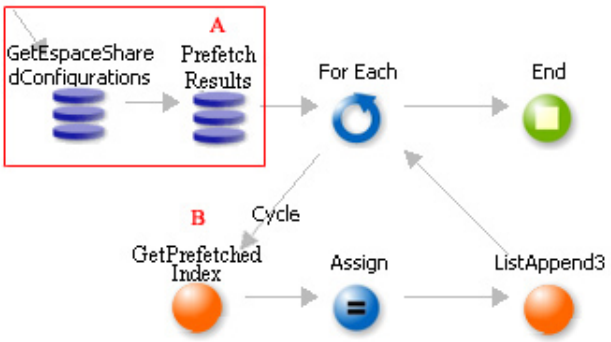

Figura 3.13: Exemplo de como internamente o padrão poderá ser restruturado

Se considerarmos que o peso das pesquisas locais têm um valor muito inferior ou nulo quando comparado com a execução de uma consulta, o ganho vai ser maior quantos mais tuplos tiver o resultado da consulta iterada. Esse ganho decresce do peso da *query* que carrega todos os valores para memória local.

Relativamente a alterações semânticas, estas acontecem quando posicionamos este padrão num cenário concorrente. Existe a possibilidade de alguns dos resultados serem diferentes devido à antecipação de todas as leituras. Neste caso é preciso fazer um levantamento de todos os cenários de utilização relevantes e decidir qual o comportamento desejado.

3.4.1.5 Eliminação de *Queries* Repetidas em ciclos

As operações de leitura que se repetem dentro de ciclos representam outro possível ponto de optimização. As leituras poderiam ser realizadas apenas uma vez no início de cada ciclo (2.1.3.1), evitando assim operações redundantes. O problema desta solução é que os dados

podem estar a ser alterados por outros utilizadores em diferentes acções. Assim, se apenas fizermos a leitura uma vez, os dados que recolhemos no início da operação correm o risco de se tornarem inconsistentes. Neste caso, a aplicação directa do algoritmo para movimentação de código invariante em ciclos iria alterar a semântica dos programas uma vez que o resultado das *queries* não é invariante.

Esta situação de aparente redundância é ainda agravada pelo facto de na plataforma, em *SQL Server*, utilizar o nível de isolamento *Read Uncommited*, que permite realizar leituras de dados inconsistentes. No caso em que existem leituras dentro de ciclos, é mais provável que se capturem resultados provisórios, mas quando esta é efectuada no início afecta todos os cálculos seguintes.

A solução deverá ter sempre em conta o que o utilizador espera como resultado final. Desta forma, podem ser exploradas alternativas que, por exemplo em vez de realizar a leitura no início, possam realizá-la temporariamente, obtendo resultados mais consistentes e leituras menos frequentes. A plataforma deverá informar o utilizador que a operação poderá beneficiar de *cache* e propor refazer a consulta em intervalos de tempo configuráveis de forma a obter resultados mais coerentes.

3.4.2 Operações de Actualização Parcial de Entidades

O padrão que foi escolhido para ser optimizado na plataforma *OutSystems* nasce da recorrente necessidade de criar formas de gerar operações de actualização parcial de entidades. As operações *Update<Entity>* actuais, recebem uma entidade e actualizam todos os valores do registo mesmo que estes não tenham sido alterados.

Por outro lado, se não for necessário actualizar todos os dados, também só é preciso carregar da base de dados aqueles que eventualmente precisem de ser lidos, por exemplo para mostrar num formulário de edição.

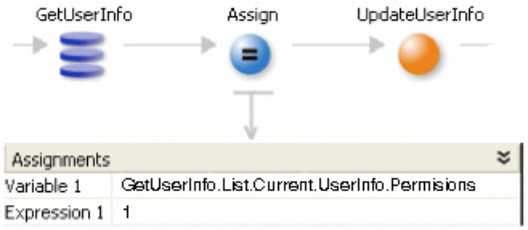

Figura 3.14: Exemplo de actualização parcial

A figura 3.14 mostra um exemplo onde é actualizado apenas um atributo de uma entidade. Com base neste exemplo, se for propagada, em tempo de compilação, a escrita sobre o identificador "Permissions"até à operação de actualização seguinte, então essa acção saberá que só

necessita de actualizar esse valor porque a informação dos restantes nunca chegou. Enriquecido com esta informação, este nó *Execute Action* poderá descartar os restantes usos de identificadores. Ao remover essa informação, o algoritmo de *Liveness* vai conseguir eliminar as leituras desnecessárias na *query* de origem.

A outra principal limitação actual, consiste na inexistência de edição concorrente de formulários. Se os mesmos dados forem lidos por diferentes utilizadores e alterados localmente, quando forem actualizados vão persistir apenas as alterações do último utilizador, que é involuntariamente responsável por eliminar as alterações do anterior.

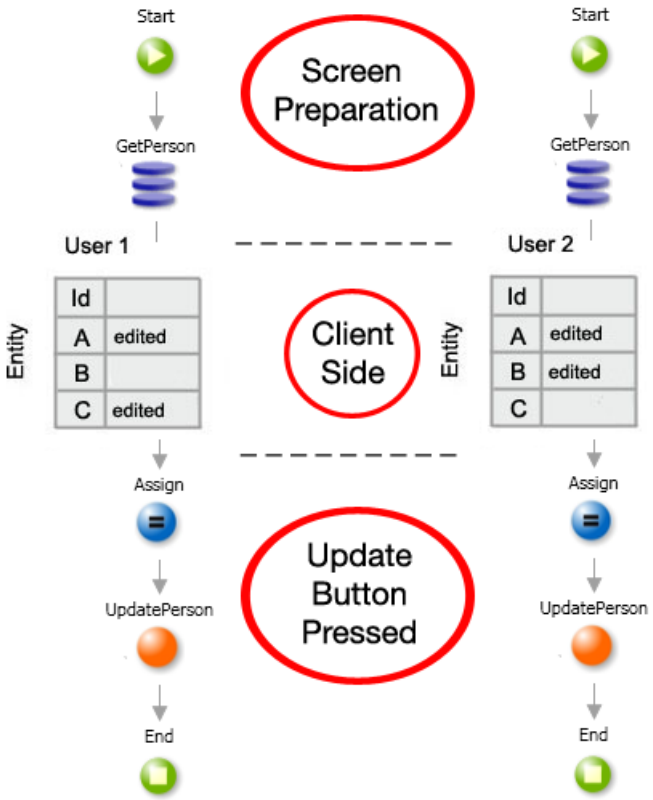

Figura 3.15: Exemplo modificação concorrente de um formulário

A figura 3.15 exemplifica a edição concorrente de uma entidade em dois formulários. No exemplo, dois utilizadores lêem os mesmos dados para um formulário (*Screen Preparation*) que depois é editado localmente (*Client Side*). Seguidamente é pressionado um botão com a respectiva afectação dos valores e actualização (*Update Button Pressed*).

Em tempo de compilação, é impossível prever quais os atributos que vão ser alterados pelos utilizadores a optimização anterior tem que propagar a escrita de todos os atributos para a

operação de *update* seguinte.

Para facilitar a união das duas modificações, é necessário introduzir optimizações em tempo de execução para verificar quais foram os atributos que realmente mudaram de valor na operação. Se no momento da afectação os atributos trocaram de valor, então devem ser enviados para o *update*.

A mesma figura mostra também que a junção dos dois resultados pode não ser simples de avaliar, porque se a operação não for disjunta existem possíveis situações de inconsistência de dados, onde porventura seria necessário detectar um conflito na escrita.

A secção 4.2 descreve o trabalho realizado no estudo de cenários e de soluções para resolver estas situações, assim como a solução implementada e validação dos resultados obtidos na optimização deste padrão.

O capítulo que se segue começa por descrever brevemente a arquitectura do compilador e do optimizador da linguagem *OutSystems*. Aqui são identificadas as suas características mais relevantes para que seja mais fácil compreender a integração da solução proposta no contexto do compilador. De seguida, são apresentadas e estudadas as diversas alternativas possíveis para resolver o problema das actualizações parciais de entidades e justificada a opção tomada. No fim, é descrita detalhadamente a implementação da solução e o processo de validação da mesma.

4

Optimização de *Queries* na Linguagem *OutSystems*

Esta secção descreve os aspectos mais relevantes do trabalho desenvolvido no âmbito desta dissertação. A primeira secção mostra a estrutura e funcionamento do compilador da linguagem *OutSytems*, com especial foco nos aspectos mais relevantes para este trabalho. O início da secção seguinte descreve o trabalho efectuado na análise e estudo de cenários de uso e das necessidades dos programadores no problema das actualizações parciais de entidades, justificando a decisão sobre qual a abordagem a seguir. Seguem-se as subsecções onde são descritos os pormenores da construção e validação da implementação desta optimização na *Agile Platform*. Por fim, são descritas de forma resumida as condições e as diferentes fases sob as quais decorreu o projecto.

4.1 Arquitectura do Compilador

Os programas definidos em *OutSystems* são normalmente constituídos por uma série de ecrãs ligados como um grafo orientado. Em cada ecrã pode existir a hipótese de transitar para outro e pode ser necessário transportar o contexto actual. Tipicamente em .NET [32] esse contexto tem o nome de *Viewstate* [33] e constitui o conjunto de valores que estão a ser manipulados pela aplicação que necessitam de transitar em cada uma destas ligações. O tamanho do *Viewstate* pode por em risco a usabilidade das aplicações, uma vez que o transporte dessa informação entre o navegador e o servidor atrasa o carregamento das páginas.

Embora todas as páginas estejam relacionadas, o compilador actua de forma completamente

independente. Cada página é percorrida individualmente e o seu código é gerado e optimizado separadamente. O mesmo acontece com as acções definidas pelo utilizador. Os procedimentos criados pelos programadores em acções separadas são também compilados e optimizados sem conhecimento do contexto de utilização, isto porque podem ser invocados em contextos completamente diferentes.

É importante referir também que nas operações de afectação todos os valores são copiados por valor. A única excepção dá-se no caso da cópia de listas de registos, onde apenas é trocado o apontador e copiado o registo corrente.

A subsecção seguinte descreve o processo de geração de código.

4.1.1 Geração de Código

Quando compiladas, as aplicações criadas em *Service Studio* podem gerar código em Microsoft .NET [32] ou Java [34], dependendo da vertente desejada. Neste trabalho foi apenas definida a geração de código em .NET porque a implementação do servidor em Java só será desenvolvida após o lançamento desta versão, por transcrição directa do código .NET. Esta transcrição não afectará de forma alguma os resultados deste trabalho, porque a semântica mantém-se nesta transcrição de código.

As linguagens utilizadas no *output* das aplicações são o C# [35, 36] para definição da lógica aplicacional e o ASP [37] para a criação das páginas *Web*. Uma das grandes vantagens implícitas a esta geração de código é que a posterior compilação das aplicações, trará uma segunda optimização realizada pelo compilador do C# ou Java.

A geração do código da lógica das aplicações passa por uma interpretação bastante directa daquele definido na linguagem *OutSystems*. Os *action flows* são percorridos de forma sequencial e cada tipo de nó tem associado um conjunto de *strings* que são escritas na altura da sua geração. Esse código geralmente depende de um conjunto de variáveis que definem os parâmetros de entrada da função ou o contexto global de execução. As acções são geradas sob a forma de função genérica invocável em diferentes contextos. Assim, os nós *Execute Action* apenas têm que escrever a chamada à função correspondente com os parâmetros correctos.

A plataforma possui ainda um conjunto de bibliotecas e código genérico de suporte às aplicações criadas.

4.1.2 Optimizador

O optimizador presente no compilador, antes de proceder a qualquer verificação, cria um grafo abstracto sobre cada um dos grafos da aplicação. Cada um dos nós apenas precisa de informação de alto nível sobre o que está a ser optimizado. Actualmente cada nó do optimizador recolhe informação sobre os identificadores que estão a ser manipulados (*use* e *def*) em cada momento de forma a computar o algoritmo de variáveis activas (2.1.1.5). Desta forma, os diferentes nós acabam por ser abrangidos por uma pequena lista de nós do optimizador, onde se destacam: *OptimizerQueryNode*, *OptimizerIteratorNode* e *OptimizerActionNode*.

Os algoritmos de transformação são aplicados sobre o grafo do optimizador que no fim da sua execução enriquece os nós referenciados com informação extra que pode ser utilizada para os optimizar no momento da escrita.

Os procedimentos definidos para acções do utilizador com as páginas Web (p.e. procedimento para tratar evento de pressionar um botão) são integrados no grafo do ecrã 4.1 e optimizados como um todo. Como já foi dito, o mesmo não acontece nas acções definidas pelo utilizador. Nestes casos a optimização é local e os nós *Execute Action* do optimizador assumem que todos os identificadores que são enviados para estas acções, acabam por ser necessários. Esta limitação leva a que em muitos casos, alguns programadores mais experientes, acabem por repetir algum código para que este seja optimizado.

A implementação da análise de variáveis activas (*Liveness analysis*) permite, através da propagação dos identificadores utilizados, reduzir o tamanho do *Viewstate* e reduzir o número valores obtidos em cada consulta SQL. Estes valores só precisam de ir para o *Viewstate* na transição de uma página se forem necessários no destino. O mesmo acontece com as consultas SQL. Neste caso, os valores só são retirados da base de dados se puderem ser lidos no futuro, assumindo todas as transições possíveis no programa. Em ambos os casos, se os atributos forem optimizados, após a leitura, estes identificadores estarão a apontar para valores por defeito, que nunca serão lidos no seu contexto.

Para além das limitações que envolvem a compilação inter-procedimental [38], o optimizador apresenta limitações ao lidar com a indexação de valores em listas. Todas as afectações sobre os registos correntes das listas, assumem que esse o registo pode mudar entre operações. Assim, esses atributos necessitam sempre de ser recolhidos da base de dados e de passar de página para página. Esta limitação limita da mesma forma as afectações sobre registos indexados, porque para efeitos de optimização, o optimizador substitui-os internamente pelo registo corrente.

O optimizador implementa ainda um algoritmo de ordenação topológica (2.1.1.4) para que o número de iterações em algoritmos de análise de fluxo dados seja mínimo.

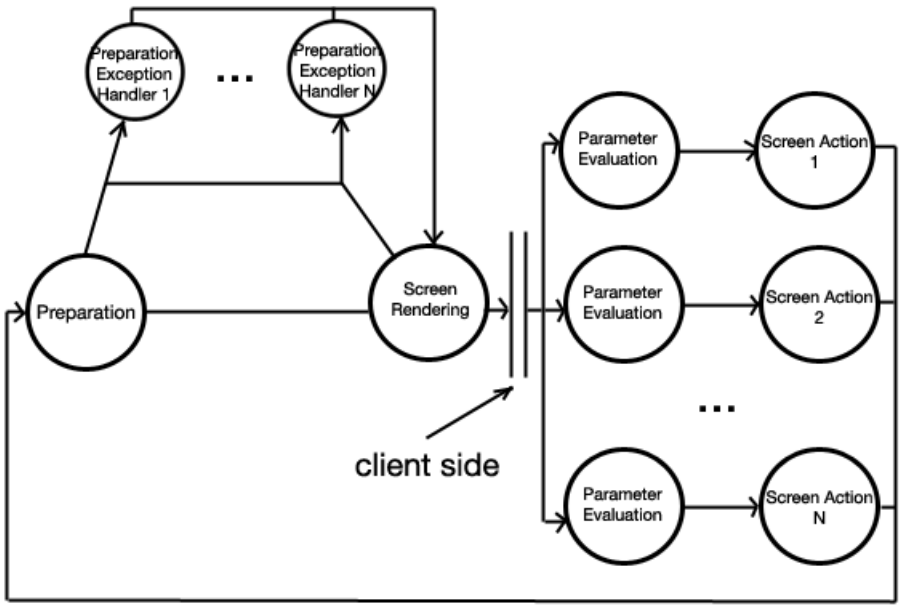

Figura 4.1: *Screen Flow Graph*

4.2 Solução

Nesta secção do documento é apresentada a solução desenvolvida para a optimização que foi proposta a resolver neste trabalho. Antes de proceder ao planeamento de uma solução e codificação, foi necessário iniciar um processo de recolha dos problemas e necessidades da versão actual da plataforma. Foi fundamental também inquirir alguns programadores da linguagem, de forma a saber qual o comportamento mais desejado e o nível de conforto perante eventuais alterações semânticas.

Uma vez tomada uma decisão, procedeu-se a uma planificação da arquitectura da solução, tendo como principal objectivo encaixar as alterações de uma forma natural no código envolvente. O propósito desta integração é tirar o maior partido das optimizações e estrutura existentes.

Seguidamente iniciou-se um processo iterativo de implementação dividido conceptualmente em duas etapas centrais: optimizações em tempo de compilação e em tempo de execução. Estas etapas foram acompanhadas do planeamento e produção constante de testes com o objectivo de garantir a consistência do produto. Para criar testes mais completos, foi inclusivamente enriquecido o mecanismo de asserções da linguagem.

Mais à frente neste documento, é apresentada também uma visão crítica sobre a solução construída, incluindo uma reflexão sobre a hipótese de alterações futuras. Como consequência

natural de um processo de optimização, é fundamental garantir que as alterações introduzidas, acabam por optimizar realmente os programas. Isto significa que é importante salvaguardar que os efeitos da complexidade da solução desenvolvida são efectivamente compensados com ganhos significativos nas aplicações desenvolvidas. Num projecto desta natureza, torna-se fácil ter acesso a aplicações reais, provando e quantificando os ganhos práticos destas alterações a vários níveis. Na parte final desta secção são mostrados e avaliados os resultados da compilação de aplicações reais e apresentada uma opinião crítica sobre esses mesmos resultados.

4.2.1 Limitações, Análise de Cenários e Decisões

Actualmente as actualizações de entidades são caracterizadas por perda de dados. As alterações feitas por utilizadores correm o risco de se perderem em *Updates* seguintes.

Existe a hipótese de recorrer à opção *GetForUpdate<Entity>* que faz um *lock* sobre a entidade na base de dados, mas o seu uso torna-se impraticável em aplicações que necessitam de ser concorrentes (p.e: preenchimento de formulários). As formas existentes de contornar este problema recorrem ao uso de *Advanced Queries*, que são de lenta concepção e propiciam o aumento de erros nas aplicações. É também necessário recorrer a *Advanced Queries* para resolver todos os tipos de conflitos entre *Updates*.

O maior problema semântico da versão actual consiste na eliminação involuntária de dados actualizados por outros utilizadores. Uma das motivações principais por trás deste projecto, consiste em facilitar a utilização de *Edit Records* com a garantia de que os utilizadores só são responsáveis pelos campos que alteram.

A figura 4.2 mostra que se dois utilizadores lerem os mesmos dados, editem os atributos localmente e seguidamente enviem as alterações, a versão que vai persistir na base de dados vai ser a última a ser enviada.

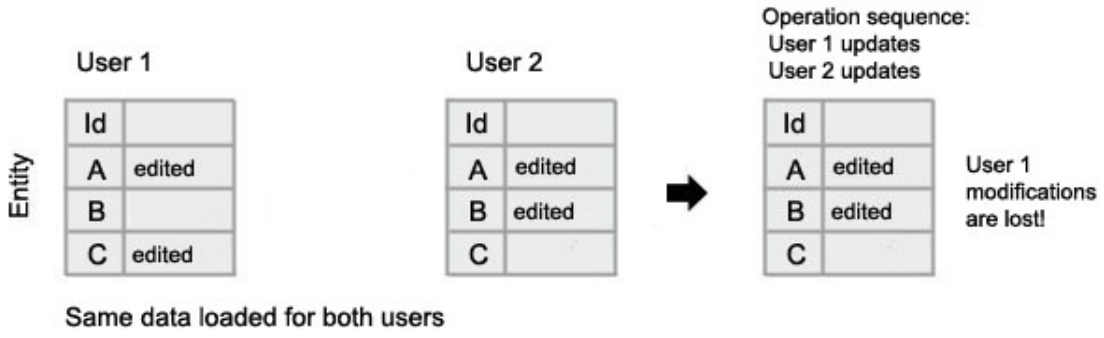

Figura 4.2: Perda de dados com actualizações sucessivas

A figura segunte (4.3) mostra que se apenas forem enviados os valores editados, as alterações do primeiro utilizador não se perdem, mas podem ter sido alteradas tendo em conta um modelo errado (não actual) da base de dados.

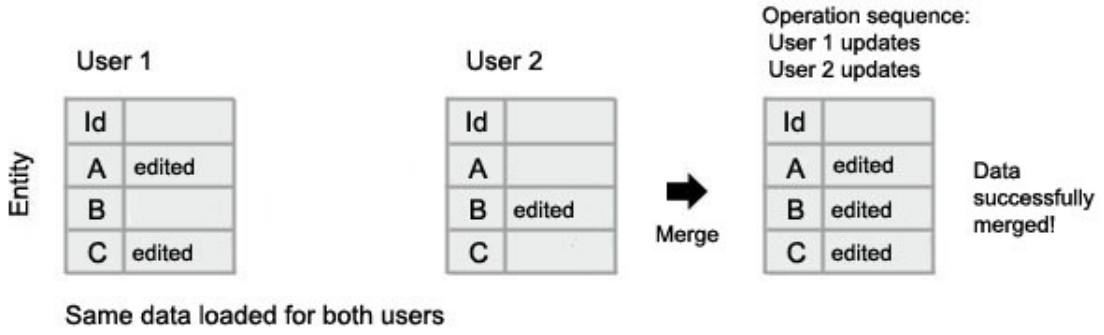

Figura 4.3: Junção de alterações bem sucedida

A outra grande preocupação consiste na actualização de atributos que foram escritos sem se verificar se entretanto esses valores mudaram na base de dados. Deveria existir um mecanismo de detecção de conflitos na actualização de valores (Figura 4.4).

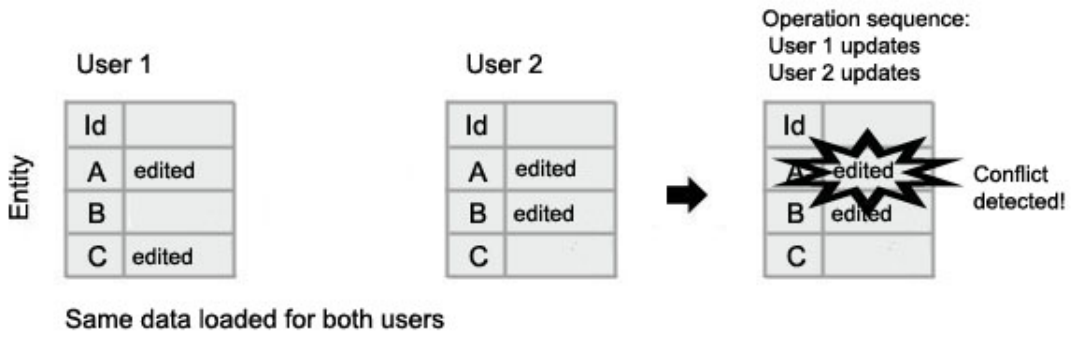

Figura 4.4: Detecção de conflito na actualização de dados

Apesar da detecção de conflitos, à primeira vista, parecer resolver o problema da inconsistência de dados provocada pelos *Updates* parciais, na realidade não é solução devido às dependências unidireccionais (Figura 4.5). Isto não é preocupante porque geralmente quem procura utilizar operações de *Update* parcial não está preocupado com a detecção de conflitos e vice-versa.

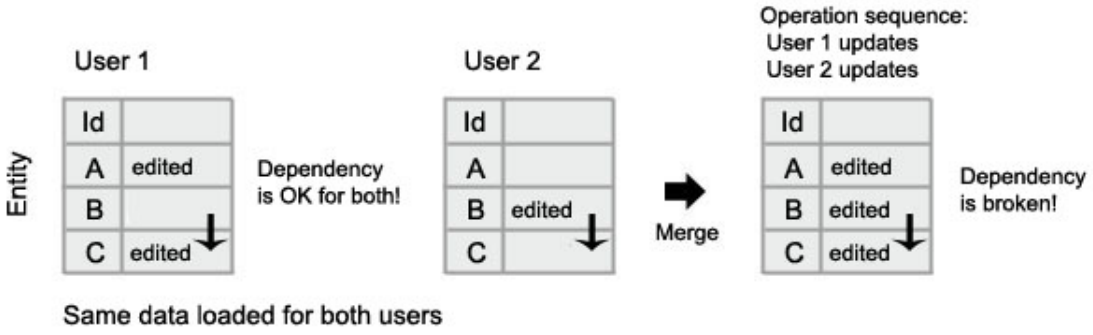

Figura 4.5: Falha de consistência na actualização de dados

A figura 4.5 mostra que se existir uma dependência unidireccional entre os possíveis valores dos atributos B e C, e for efectuada uma união das alterações, essa união pode resultar num resultado inconsistente. Imaginemos um exemplo simplificado com o par (Brasil, Dollar) lido por dois utilizadores. Se os utilizadores alterarem os dados para os valores (USA, Dollar) e (Brasil, Real) respectivamente, ambos os cenários são válidos localmente mas a união das modificações (USA, Real) é inválida devido à dependência da moeda em relação ao país.

Análise de Cenários

Como foi mencionado anteriormente, antes de projectar a solução para a optimização geral que foi proposta a resolver, tornou-se fundamental recolher informação sobre quais são as necessidades reais dos programadores e identificar os problemas da versão actual da plataforma *OutSystems*.

A recolha desta informação foi compilada num conjunto de cenários representativos de situações suboptimizadas, comportamentos indesejáveis ou soluções complicadas para os programadores.

Alguns dos cenários de utilização escolhidos foram:

1. Dois utilizadores modificam a mesma entidade no mesmo formulário (os mesmos atributos incluindo o *last modified*)

2. Dois utilizadores modificam a mesma entidade em formulários diferentes (atributos diferentes incluindo o *last modified*)

3. Um utilizador grava uma entidade num formulário carregado com informação muito antiga (p.e. deixado aberto durante a noite). Entretanto a entidade foi modificada por um processo.

4. Um formulário é usado para editar uma entidade com um atributo de dados binários de grandes dimensões.

5. Uma acção é usada para actualizar uma entidade que contem um atributo binário de grandes dimensões.

6. Cenário de uma empresa de seguros: *Advaced Queries* são utilizadas, em vez de *Update<Entity>*, para suportar padrões de detecção de conflitos com base em campos *last modified*

7. Cenário de uma empresa de viagens: A maioria dos formulários requerem um *Get<Entity>* e uma afectação antes de cada operação de *Update<Entity>* para garantir que apenas os campos editados são actualizados na base de dados. As entidades são muito grandes e alteradas em múltiplos ecrãs com formulários.

8. Cenário de uma empresa de energia: Entidades contêm atributos de soma. Quando são actualizados os valores da entidade, o valor total é calculado e a entidade actualizada na base de dados. Dois utilizadores lêem os mesmos dados, o primeiro actualiza um atributo e o respectivo total e o segundo utilizador faz o mesmo, modificando um atributo diferente.

9. Um formulário é validado tendo em conta valores antigos (no *Viewstate*) antes da entidade ser enviada para a base de dados.

Cenário	Limitações actuais
1	Sem detecção de conflitos. As primeiras alterações perdem-se.
2	As alterações do primeiro utilizador perdem-se no segundo *Update<Entity>*.
3	Sem detecção de conflitos. As alterações do processo perdem-se.
4	O tamanho do *Viewstate* é demasiado grande.
5	As operações para obter e actualizar entidades são lentas.
6	As *Advanced Queries* têm que ser editadas manualmente quando há uma alteração na entidade. Maior probabilidade de erros.
7	Trabalho aborrecido e requer que a consistência dos valores do *Edit Record* seja verificada manualmente.
8	Sem detecção de conflitos. As primeiras alterações perdem-se.
9	Os dados anteriores perdem-se, mas não há problemas de consistência dos dados.

Tabela 4.1: Listagem das limitações actuais

Listagem de Possíveis Soluções

De acordo com as limitações da versão actual da plataforma, foram pensadas e debatidas seis alternativas para solucionar este problema que tentam, pelo menos, eliminar parte dos falhas da versão anterior. Estas soluções têm como objectivo alterar a forma como as operações de actualização de entidades são realizadas actualmente: todos os valores da entidade são lidos e escritos, independentemente de terem sido alterados ou de entretanto alguém os ter modificado.

Existem outras soluções possíveis para além das que vão ser apresentadas, mas num produto desta natureza, é essencial garantir a simplicidade e clareza das operações. Soluções muito complexas, mesmo que sejam completas, são inconvenientes porque se tornam difíceis de explicar e podem induzir os programadores a erros graves na sua utilização.

As propostas debatidas com os responsáveis pelo produto foram as seguintes:

Actualização de valores escritos e lidos: Esta solução consiste em enviar para a base de dados todos os valores que foram escritos (não necessariamente modificados) e todos os que necessitam de ser lidos: marcados com "used". Este comportamento, por ser único, introduz necessariamente uma mudança semântica.

Comportamento definido na entidade (total/modificados): Todas as operações de *Update<Entity>* são coerentes com aquilo que foi definido na entidade. É possível optar entre o comportamento actual (actualizar todos) ou por apenas actualizar os atributos que foram modificados (o valor realmente mudou em relação ao que estava na base de dados).

Comportamento definido na entidade (total/total com verificação conflitos/modificados): Idêntico ao anterior, mas permite ao programador definir se pretende que haja verificação de conflitos ao nível de todos os atributos da entidade. Uma situação de conflito significa que um atributo, que foi enviado para a base de dados, encontrou lá um valor diferente do que tinha quando foi lido (alguém o alterou entretanto).

Comportamento definido em cada operação de *Update<Entity>* (total/modificados): É possível parametrizar todas as operações de actualização de entidades de acordo com as necessidades do programador em cada ponto do código. O programador pode decidir manter o comportamento antigo ou actualizar apenas os atributos modificados.

Actualizar apenas atributos modificados: Todas as operações de actualização de entidades são coerentes e apenas actualizam os atributos da entidade que mudaram de valor. Este comportamento, por ser único, introduz necessariamente uma mudança semântica.

Actualizar apenas atributos modificados com verificação de conflitos ao nível dos atributos: Semelhante ao anterior, mas permite ao programador especificar se pretende que haja de controlo de conflitos ao nível dos atributos de cada entidade. O programador precisa de especificar quais os atributos de cada entidade onde pretende verificar se há conflitos no momento da escrita na base de dados.

Depois de feito o levantamento das soluções realizou-se um enquadramento destas na resolução dos cenários indicados anteriormente. O resultado foi o seguinte:

i) Actualização de valores escritos e lidos

ii) Comportamento definido na entidade (total / modificados)

iii) Comportamento definido na entidade (total / total com verificação conflitos / modificados)

iv) Comportamento definido em cada operação de *Update<Entity>* (total / modificados)

v) Actualizar apenas atributos modificados

vi) Actualizar apenas atributos modificados com verificação de conflitos ao nível dos atributos

	Versão →	actual	i)	ii)	iii)	iv)	v)	vi)
Cenários de utilização	1	Dados perdidos	Sem detecção de conflitos	Sem detecção de conflitos	Ok	Sem detecção de conflitos	Sem detecção de conflitos	Ok
	2	Dados perdidos	Ok	Dados perdidos / Ok	Ok	Dados perdidos / Ok	Ok	Ok
	3	Dados perdidos	Possível perda de dados	Dados perdidos / Ok	Ok	Dados perdidos / Ok	Ok	Ok
	4	Lento	Ok	Ok	Ok	Lento / Ok	Ok	Ok
	5	Lento	Ok	Ok	Ok	Ok	Ok	Ok
	6	Complexo	Complexo	Complexo	Ok	Complexo	Complexo	Ok
	7	Complexo	Ok	Ok	Ok	Ok	Ok	Ok
	8	Dados perdidos	Ok	Dados perdidos / Dados inconsistentes	Ok	Dados perdidos / Dados inconsistentes	Dados inconsistentes	Ok
	9	Dados perdidos	Ok / Possível perda de dados	Dados perdidos / Dados inconsistentes	Ok	Dados perdidos / Dados inconsistentes	Dados inconsistentes	Dados inconsistentes
	Design	-	Difícil de explicar	Updates não configuráveis	Ok	Inconsistência entre Updates	Alteração crítica	Exigente
	Custo	-	Aceitável	Aceitável	Elevado	Aceitável	Aceitável	Muito Elevado

Tabela 4.2: Avaliação dos cenários

i) Actualização de valores escritos e lidos:

- Esta abordagem resolve apenas parte do problema de actualizações desnecessárias, porque não verifica se os valores mudaram realmente em tempo de execução. Como actualiza os atributos usados para leitura e todos os que foram escritos, previne inconsistências de valores, mas não elimina o problema grave de se perderem alterações de outros utilizadores. A alteração introduz uma mudança semântica que, por si só, já é difícil de clarificar e ainda implica que em cada *Update* os valores a actualizar sejam validados pelo programador. Não resolve o problema da detecção de conflitos.

ii) Comportamento definido na entidade (total / modificados)

- A opção de ter uma propriedade nas entidades que define o comportamento das operações de *Update* permite que as operações sobre a mesma entidade sejam coerentes entre si e que seja mais fácil modificar futuramente o seu valor. A opção de manter o comportamento antigo permite uma transição mais suave para os programadores e uma manutenção da semântica dos programas que foram desenvolvidos em versões anteriores da plataforma. As operações de *Update* parcial podem criar situações de inconsistência de dados em entidades com dependências internas de valores, mas neste caso deve ser seleccionada a outra opção. Esta vertente não resolve o problema de detecção de conflitos.

iii) Comportamento definido na entidade (total / total com verificação conflitos / modificados)

- Esta solução resolve os problemas da anterior, porque introduz a vertente de detecção de conflitos. No entanto, esta solução tem um custo mais elevado de implementação e desenho.

iv) Comportamento definido em cada operação de *Update*<*Entity*> (total / modificados)

- Esta solução é em tudo semelhante à apresentada em ii) mas permite a flexibilidade de controlo ao nível de cada *Update*. Apesar da vantagem em cada operação ser configurável, perde-se coerência entre operações e torna-se muito pouco prático alterar posteriormente a propriedade em todos os pontos do código, conduzindo a problemas de usabilidade.

v) Actualizar apenas atributos modificados

- Optar por actualizar apenas os campos que foram modificados, introduz uma mudança semântica forte na linguagem. Existe sempre o perigo de se esperar o comportamento antigo e de as aplicações falharem inesperadamente. Desta forma, não existe forma prática de adoptar o comportamento anterior. Esta via não resolve o problema de detecção de conflitos e pode sempre levar a estados inconsistentes em entidades com interdependência entre atributos.

vi) Actualizar apenas atributos modificados com verificação de conflitos ao nível dos atributos

- A introdução de detecção de conflitos à solução anterior vai resolver a maior parte dos problemas de consistência de dados. Mas algumas situações com dependência de valores como a identificada em 4.5 vão continuar a existir. Esta solução tem um custo bastante mais elevado de implementação e desenho.

Decisão

Depois do levantamento destas opções e das implicações de cada uma, foi decidido avançar para a solução descrita em ii). A decisão teve como base as limitações de tempo até à saída da versão seguinte e também a vantagem de poder replicar o comportamento antigo, garantindo uma transição natural para quem programava na versão anterior. Com esta solução, torna-se fácil também avançar para a solução iii) sem perder o trabalho anteriormente realizado.

As questões de usabilidade também tiveram um peso importante. Existe sempre alguma reserva para modificações que impliquem alterar a interface das operações, particularmente no que leva à adição de novas propriedades.

Não foi adoptada a solução descrita em iv), devido aos problemas de inconsistência e questões de usabilidade. Já em relação às soluções descritas em iii) e vi), a opção deve-se principalmente com questões de tempo derivadas da complexidade deste tipo de solução. As restantes opções introduziam mudanças semânticas fortes e não aceitáveis, que por exemplo no caso i), implicavam a atenção redobrada por parte dos programadores.

Inquéritos aos Programadores

Na sequência deste projecto, foram também inquiridos seis programadores experientes da linguagem, com o objectivo de saber até que ponto a opção de *Update* parcial é útil para as aplicações. Estes inquéritos tiveram a intenção de estudar tanto o impacto no trabalho que foi realizado até à altura como no trabalho futuro de desenvolvimento.

Os resultados revelaram que 100% os inquiridos conhecem o comportamento actual da operação de *Update<Entity>* e reconhecem que o comportamento desejado passaria por actualizar apenas os valores que foram alterados. Após a apresentação de um cenário onde os dados poderiam ficar inconsistentes, um dos entrevistados mudou de opinião passando e esta última passaria para pouco mais de 80%.

Apesar disto, os resultados mostraram que em 19% dos casos da utilização de *Updates* os programadores quereriam evitar consequências da actualização parcial, devido à possibilidade de provocar inconsistência de dados. Por outro lado, os mesmos pensam que em 79% dos casos seria vantajoso o seu uso e desejariam mesmo ter a possibilidade de o utilizar (mesmo que não fosse o comportamento por omissão).

A avaliação destes inquéritos reforçou a necessidade de avançar com esta optimização e de manter o comportamento anterior.

4.2.2 Descrição da Solução

A solução apresentada para optimização de operações de actualização de entidades está dividida em duas fases. A primeira introduz alterações em tempo de compilação para reduzir o fluxo de informação entre as bases de dados e as aplicações, através da aplicação de um algoritmo de análise fluxo de dados. A fase seguinte enriquece as aplicações com informação dos atributos que trocaram de valor em tempo de execução. Esta última, permite reduzir ainda mais o conjunto de valores enviado em cada operação de actualização.

Em ambos os casos, a informação dos atributos que são necessários nas operações de actualização, tem que passar por parâmetro para estas acções. Na versão actual do compilador, nenhuma *Entity Action* é optimizada de acordo com o contexto em que está inserida, sendo no entanto gerado código específico para cada entidade diferente. Assim, o código é gerado de forma genérica para qualquer operação sobre a entidade, numa única função (por exemplo *Update<Person>*), que pode ser chamada em diferentes contextos. Esta decisão, para além de evitar código repetido e de possibilitar juntar todas as *EntityActions* num único ficheiro fonte, permite ganhos com compilação parcial, porque este código raramente precisa de ser recompilado (a não ser que o modelo de dados se altere).

Sendo o *Update<Entity>* uma *Entity Action*, a sua compilação gera uma função que recebe os dados da entidade e a actualiza na base de dados através de um comando SQL. A informação sobre quais os atributos a actualizar é passada para este código genérico, garantindo que apenas aqueles que foram modificados é que vão ser actualizados.

Essa informação é expressa num *array* de *bits*, cujo conteúdo é calculado inicialmente apenas com informação recolhida em tempo de compilação e numa segunda fase com informação de *runtime*. Esta solução parece adequada visto que queremos combinar a análise de informação estática com dados recolhidos em tempo de execução para maximizar os ganhos finas.

As secções seguintes descrevem a implementação destas duas fases e as alterações que foi necessário introduzir no processo de geração de código do compilador *OutSystems* para as suportar.

4.2.2.1 Optimizações em Tempo de Compilação

A ideia geral da nossa solução consiste na propagação, pelo grafo do optimizador, da informação dos atributos que são afectados desde a leitura de uma entidade da base de dados até à próxima operação de actualização dessa entidade. Com esta informação, cada nó de *Update* consegue saber quais os campos que podem vir a mudar de valor (porque não é certo que mudem para um valor diferente), e assim retirar o uso desses atributos que nunca serão alterados nessa operação.

Actualmente, no grafo do optimizador, os nós de *Update<Entity>* são categorizados como

OptimizerActionNodes, que marcam o uso de todos os campos da entidade que é enviada como parâmetro da acção. Isto significa que se a entidade foi capturada por uma *query*, então grande parte dos campos desse *Entity Record*, tiveram que ser recolhidos da base de dados antes de o *Update* ser realizado.

O algoritmo de *Liveness* implementado no optimizador permite evitar algumas leituras desnecessárias, caso se detecte que um valor vai ser redefinido antes do seu uso. Nesta situação as *queries* são ligeiramente optimizadas, mas sem a optimização dos *updates* continuam a precisar de obter valores da base de dados que possivelmente nunca vão ser utilizados, a não ser numa actualização para o mesmo valor.

Ao remover os identificadores da lista *use* dos nós de *Update*, os campos que eram lidos das *queries* apenas para ser usados na actualização, vão ser optimizados pelo algoritmo de *Liveness* e consequentemente retirados dessas *queries*. A única excepção aqui aplica-se à chave da entidade, que é sempre necessária para a operação de actualização (para a condição *where* do *SQL*).

Propagação de Atributos Escritos

Para que se consiga propagar convenientemente o conjunto de atributos que são alvo de operações de atribuição é preciso guardar, a informação dos identificadores que foram marcados como *changed* à entrada (*ChangesIn*) e à saída (*ChangesOut*) de cada nó (*N*). Isto significa que cada ponto do grafo vai receber a informação dos valores que foram marcados como *changed* anteriormente e tem também a capacidade de modificar essa lista (ver secção 2.1.1.2).

O algoritmo para propagação de atributos alvo afectações consiste na iteração sucessiva dos nós do grafo, onde cada um destes nós é responsável por recolher, a cada iteração, o conjunto dos atributos modificados nos nós predecessores. Este método é repetido até se atingir um ponto onde nenhum dos nós alterou as suas listas de valores *changed* (ponto fixo - 2.1.1.3). Em cada nó do grafo vão ser aplicadas as seguintes regras:

$$ChangesIn(N) = \cup_{\,S \text{ is predecessor}(N)} \left(ChangesOut(S)\right)$$

$$ChangesOut(N) = gen(N) \cup \left(InChanged(N) - kill(N)\right)$$

$$kill(N) = \begin{cases} N \text{ is } Update \rightarrow use(N) \cap ChangesIn(N) \\ else \qquad\quad \rightarrow def(N) \end{cases}$$

$$gen(N) = \begin{cases} N \text{ is } Query \rightarrow \emptyset \\ else \qquad\quad \rightarrow \left(\cup_{\,x \text{ is assigned in } N} \left(\{x\}\right)\right) \cup \left(\cup_{\,y \text{ is exported to scope in } N} \left(\{y\}\right)\right) \end{cases}$$

• **Funções *ChangesIn(N)* e *ChangesOut(N)*:**

No grafo do optimizador, cada um dos nós define duas listas de identificadores com o mesmo nome das funções geradoras: *ChangedIn* e *ChangedOut*. A lista *ChangesIn* é resultado da união de todas as listas *ChangesOut* dos nós predecessores, dando origem à propagação no sentido do fluxo de execução. Cada um dos nós pode depois alterar a sua lista *ChangesIn*, introduzindo novos identificadores com a função *gen* ou eliminando alguns dessa lista com a função (*kill*), definindo a lista de saída *ChangesOut*.

• **Função *gen(N)*:**

A geração de novos valores para a lista de saída (*gen*), provém essencialmente dos nós de afectação que propagam o identificador do lado esquerdo dessa operação ($left := right$). Neste contexto interessa apenas propagar identificadores de atributos de entidades que possuam a propriedade de *Update* parcial.

As afectações podem ser de diferentes tipos:

- Afectação directa a um atributo de uma entidade:

  O identificador do lado esquerdo é adicionado à lista *gen.*

- Afectação a um *Record*, *Entity Record* ou *Record List*:

  Se o lado direito da afectação for uma expressão de identificador (*IdentifierExpression*), então expande-se esse identificador (de forma a mostrar todos os sub-identificadores do registo ou da lista) e percorrem-se esses identificadores verificando se cada um deles está presente na lista *ChangesIn*. O identificador correspondente do lado esquerdo só é inserido na lista *gen* se o correspondente do lado direito for encontrado, ou seja, se esse atributo foi anteriormente alvo de uma afectação. Se o atributo não for encontrado na lista, o correspondente do lado esquerdo é removido da lista *gen*, para garantir que a informação dos valores alterados é mantida quando ocorrem cópias de registos ou listas.

  No caso oposto, torna-se impossível expandir lado direito da afectação e adicionam-se todos os identificadores do lado esquerdo à lista *gen*.

No entanto, outros nós são responsáveis pela inserção de variáveis como *changed* no *scope* do *action flow* corrente. Isto acontece porque, quando são chamadas acções ou funções que retornam tipos *Entity Record* (ou outros que os possam suportar: listas, registos ou estruturas), não existe a informação se esses valores vêm ou não alterados do outro lado, assim assume-se o pior caso, ou seja, que vêm todos alterados.

A única excepção a esta regra são as *queries*, porque apesar de exportarem variáveis para o *scope*, os seus valores representam o estado da base de dados e por isso não necessitam de ser marcados como *changed*.

• **Função *kill(N)*:**

A eliminação de valores da lista de entrada (*kill*) dá-se em duas situações. A primeira surge quando os identificadores que entram na lista *ChangesIn* estão contidos na lista local de definições (*def*). Se os valores são novamente definidos, então deixam de estar marcados como alterados. A segunda situação acontece nos nós de *Update*, que retiram os identificadores da lista porque estes passam a estar coerentes com a base de dados.

Terminação do Algoritmo

Para efeitos de correcção vamos mostrar que o algoritmo do ponto fixo implementado neste trabalho termina sempre.

Aplicando iterativamente as regras definidas anteriormente ao conjunto de todos os nós, as listas *ChangesIn* e *ChangesOut* vão crescer a cada iteração até se estabilizarem num ponto em que todos os identificadores em cada nó são iguais aos da última iteração (ponto fixo). Por definição (2.1.1.3), só se garante a terminação deste algoritmo se o conjunto de identificadores for finito e se as funções geradoras para cada nó forem monótonas. Como a informação é estática o conjunto de identificadores está sempre bem definido e é finito. Só precisamos de garantir que as funções geradoras são monótonas.

Uma função f é monótona sob o conjunto de identificadores (conjunto de ordem parcial $(\varphi, \sqsubseteq)$) se:

- Seja: φ um conjunto de identificadores

- Então: $\forall\ \alpha,\ \beta \in \varphi : \alpha \sqsubseteq \beta \Rightarrow \mathrm{f}(\alpha) \sqsubseteq\ \mathrm{f}(\beta)$

Unindo as duas formulas, temos:

$$ChangesOut(N) = gen(N)\ \cup\ ((\cup_{\ S\ is\ predecessor(N)}\ (ChangesOut(S))) - kill(N))$$

$$ChangesOut(N) = f(\cup_{\ S\ is\ predecessor(N)}\ (ChangesOut(S)))$$

Hipótese: *ChangesOut(N)* só cresce

i) Provar (1ª iteração):
ChangesIn(S) = $\emptyset$
$\Rightarrow$ ChangesOut(N) = gen(N) $\cup$ ($\emptyset$ - kill(N)) = gen(N)
$\Rightarrow$ ChangesOut(N) $\geq \emptyset$
(nota: o conjunto de valores exportados para o *scope* é constante.)

ii) Provar (nª iteração):
Assumindo que *ChangesOut(S)* aumenta em alguns predecessores, então:

$$\cup_{S \text{ is } predecessor(N)} (ChangesOut(S))$$

só pode crescer ou manter-se igual depois da iteração. O mesmo acontece com:

$$gen(N) \cup ((\cup_{S \text{ is } predecessor(N)} (ChangesOut(S)) - kill(N))$$

porque *gen(N)* é um conjunto constante e porque expandindo o restante da formula para as duas hipóteses possíveis de *kill(N)*:

$$ChangesIn(N) - (ChangesIn(N) \cap use(N)) \Rightarrow ChangesIn(N) - use(N)$$
$$ChangesIn(N) - def(N)$$

como *def(N)* e *use(N)* são constantes, e *ChangesIn(N)* é crescente (por hipótese), as duas possibilidades são também crescentes. Isto implica que a função *ChangesOut(N)* seja monótona crescente e consequentemente *ChangesIn(N)* também o é.

Algoritmo para Propagação de Atributos Escritos

A listagem de código seguinte (4.2.1) define de forma abstracta o algoritmo para propagação de atributos alvo de afectação, que pode correr sob qualquer grafo de um *action flow*. Antes da execução deste algoritmo, é realizada a operação de ordenação topológica, implementada no compilador, para que o número de iterações até que o algoritmo estabilize seja o mínimo.

A função recebe o conjunto dos nós do grafo (do optimizador) de um *action flow* e itera essa lista de nós continuamente, aplicando as funções definidas anteriormente, até que a propagação de atributos estabilize no ponto fixo.

Após a execução deste algoritmo, cada nó de *Update* de cada *action flow* vai estar enriquecido com informação suficiente para verificar quais são os campos que são realmente necessários à operação. Desta forma, a última instrução do algoritmo é à chamada da função que corta os usos dos nós de *Update<Entity>*, explicada na secção seguinte.

Algorithm 4.2.1: CALCULATE CHANGES PROCEDURE(*pseudocodigo*)

```
procedure CALCULATECHANGES(graphNodes)
changes ← true
while changes
do { changes ← false
     for each node ∈ graphNodes
     do { gen ← {}
          kill ← {}
          for each preNode ∈ node.PreviousNodes
          do { clone ← node.ChangesOut
               node.ChangesIn ← node.ChangesIn ∪ (preNode.ChangesOut)
               if clone ≠ node.ChangesIn
                 then changes ← true
          kill ← kill ∪ node.def
          if node is OptimizerUpdateNode
            then kill ← kill ∪ (node.use ∩ node.ChangesIn)
          if node.exportsToScope ≠ null and not node is Query
            then { for each id ∈ node.exportsToScope
                   do gen ← gen ∪ id.ExpandedIdentifiers
          if node.assigment ≠ null
            then { leftside ← node.assignment.leftside
                   rightside ← node.assignment.rightside
                   if leftside is {Record or RecordList or EntityRecord}
                     then { if rightside is IdentifierExpression
                            then { for i ← 0 to rightside.ExpandedIdentifiers.Count − 1
                                   do { rightId ← rightside.ExpandedIdentifiers.Get(i)
                                        leftId ← leftside.ExpandedIdentifiers.Get(i)
                                        if node.ChangesIn.Contains(rightId)
                                          then gen ← gen ∪ {rightId}
                                          else gen ← gen − {leftId}
                            else for each id ∈ leftside.ExpandedIdentifiers
                                 do gen ← gen ∪ {id}
                     else gen ← gen ∪ {leftside}
          clone ← node.ChangesOut
          node.ChangesOut ← gen ∪ (node.ChangesIn − kill)
          if clone ≠ node.ChangesOut
            then changes ← true
OptimizeUpdates(GraphNodes);
```

Algoritmo para Corte de Usos de Updates

O algoritmo seguinte (4.2.2) é responsável por percorrer cada *update*, verificando se se trata de uma operação total ou parcial. No caso de se tratar de um *update* parcial, procede-se à eliminação de atributos desnecessários. Os identificadores de cada entidade não podem ser removidos do nó, porque são necessários para a encontrar na base de dados. Por outro lado, se o identificador da entidade for alterado, procede-se a uma operação de *update* total porque a entidade referida passou a ser outra. Este método pode ser utilizado, por exemplo, para copia de entidades.

Algorithm 4.2.2: CUT CHANGES FROM UPDATES(*pseudocodigo*)

```
procedure OPTIMIZE UPDATES(graphNodes)
for each node ∈ graphNodes.GetUpdateNodes
do { changedId ← false
     remove ← {}
     if not node.IsFullUpdate
       then { for each id ∈ node.use
              do { attr ← id.GetReferencedObject
                   isKey ← attr.IsPrimaryKey or attr.IsTenantId
                   isEntityPrimaryKey ← attr.IsPrimaryKey
                   if not node.ChangesIn.Contains(attr)
                     then { comment: the Primary Key is used in the where condition
                            if not isEntityPrimaryKey
                              then remove ← remove ∪ {attr} }
                     else { comment: any Key modification results in a full update
                            if isKey
                              then { changedId ← true
                                     break } }
              if changedId
                then remove ← {} }
     node.use ← node.use − remove
     comment: by now all updates have the minimum "uses"
```

Após a execução deste algoritmo, espera-se que tenha sido removida uma quantidade considerável de usos nas acções de *Update*. Isso significa que alguns valores que tinham sido

lidos da base de dados e que, através da lógica aplicacional, nunca poderiam ter sido alterados pelo programa, não necessitam de ser trazidos da base de dados. O algoritmo de *liveness* existente actualmente no optimizador encarrega-se encontrar estas situações, aumentando a sua capacidade de optimização anterior. Esse algoritmo é chamado logo após a terminação deste último, percorrendo os nós no sentido inverso da ordenação topológica, porque se trata de uma propagação de valores no sentido inverso ao da execução.

Alterações na Geração de Código

Depois de implementados os algoritmos anteriores, foi necessário alterar tanto o código gerado das operações *Update<Entity>* como das *Execute Action* que invocam as anteriores, para fazer uso destas optimizações. Em termos de código, os nós *Execute Action* apenas geram chamadas aos métodos genéricos *Update<Entity>*, enviando a entidade a actualizar como parâmetro. Assim, junta-se à chamada deste método, a informação de quais foram os campos que possivelmente trocaram de valor (resultado do algoritmo 4.2.2), enviando um vector de *bits*. Dentro de cada *Update<Entity>*, a criação da consulta *SQL* tem que ter em conta os valores deste novo parâmetro.

O exemplo seguinte (4.1) exemplifica a geração de código de uma destas operações e o local onde é passado o *BitArray*.

Listagem 4.1: Exemplo de geração de código *Execute Action* de um *Update<Entity>*

```
1 ExtendedActions.UpdatePerson(heContext, /* dump BitArray here */,
2                   localVars.queryResQuery1_outParamList.CurrentRec);
```

A listagem seguinte (4.2) representa o exemplo de código (genérico) gerado para uma operação *update* de uma entidade *Person* com 3 atributos (Id, Username e Password). Dentro dessa função é criada a *string* SQL com o comando de actualização.

Existem ainda dois pormenores que necessitam de ser mencionados. Se for passado por argumento um valor *null* em vez do *BitArray*, a operação reage como se tivesse o comportamento antigo, ou seja, actualiza todos os valores (linha 6). Esta vertente é apenas utilizada em situações onde as entidades que apenas suportem *updates* totais. O outro pormenor consiste na existência de uma operação de actualização que não altera nenhum valor (linha 36). Isto acontece quando nenhum dos campos foi alterado, e a necessidade desta operação deve-se à geração de um evento de *update* que é essencial para garantir os efeitos laterais.

Internamente a esta função o *BitArray* tem o nome *usedFields*, representado os campos que vão ser usados na operação de actualização.

Listagem 4.2: Exemplo de geração de código da *Extended Action Update<Entity>*

```
1  public static void UpdatePerson(HeContext heContext, BitArray usedFields,
2                       RCPersonRecord inParamSource) {
3
4     ENPersonEntityRecord ssENPerson = inParamSource;
5     IDbTransaction trans = DBTransactionManager.Current.GetMainTransaction();
6     if(usedFields == null) {
7        usedFields = new BitArray(new bool[3] { true, true, true });
8     }
9
10    string updateSet = "SET NOCOUNT OFF;UPDATE osusr_2zk_Person SET ";
11
12    string parameters = "";
13    parameters = (usedFields[1]? (parameters.SuffixIfNotEmpty(", ") +
14             "\"USERNAME\" = @ssUsername"): parameters);
15    parameters = (usedFields[2]? (parameters.SuffixIfNotEmpty(", ") +
16             "\"PASSWORD\" = @ssPassword"): parameters);
17
18    string whereCondition = " WHERE \"ID\" = @ssENPersonssId";
19
20    string sql = updateSet + parameters + whereCondition;
21
22    IDbCommand sqlCmd = DBHelper.CreateCommand(trans, sql);
23    int counter = 0;
24    if(usedFields[1]) {
25    DBHelper.AddParameter(sqlCmd, "@ssUsername", DbType.String,
26       ssENPerson.ssUsername);
27    counter++;
28    }
29    if(usedFields[2]) {
30    DBHelper.AddParameter(sqlCmd, "@ssPassword", DbType.String,
31       ssENPerson.ssPassword);
32    counter++;
33    }
34    DBHelper.AddParameter(sqlCmd, "@ssENPersonssId", DbType.Int32,
35       ssENPerson.ssId);
36    if(counter == 0) {
37       String dummyUpdate = "\"USERNAME\" = \"USERNAME\"";
38       String noUpdate = updateSet + dummyUpdate + whereCondition;
39       sqlCmd.CommandText = noUpdate;
40    }
41    if(DBHelper.ExecuteNonQueryDontCleanParameters(sqlCmd,
42       "Entity Action UpdatePerson", true) <= 0){
43       throw new DataBaseException("...");
44    }
45    DBTransactionManager.Current.ReleaseTransaction(trans);
46 }
```

Uma vez conjugados todos estes factores, as actualizações de valores na base de dados estão optimizadas, recorrendo apenas a informação estática.

4.2.2.2 Optimizações em Tempo de Execução

A solução anterior permite reduzir a quantidade de informação enviada para as operações de actualização, mas não garante que os valores que são enviados na acção sejam diferentes daqueles que foram lidos. O caso mais típico onde esta optimização falha é na alteração de formulários. Todos os campos precisam de ser lidos e mostrados aos utilizadores e, na submissão do formulário, todos os atributos são afectados, porque não se consegue saber em tempo de compilação quais foram alterados. Isto deve-se tanto ao desconhecimento da informação que está na base de dados como ao comportamento imprevisível dos utilizadores. Torna-se então impossível obter esta informação sem recorrer a mecanismos que, em tempo de execução, verifiquem se os atributos alvo de afectações trocaram de valor quando manipulados pelos utilizadores ou pela aplicação.

A segunda parte da solução visa colmatar este problema. Cada *Entity Record* precisa de guardar informação de quais foram os campos que alteraram o seu valor em tempo de execução, para enviar esta informação posteriormente às operações *Update<Entity>* que na versão anterior recebiam informação estática.

Uma solução baseada em informação gerada em tempo de execução não necessita de descartar por completo a solução anterior, e pode até ganhar com a sua existência. Neste caso concreto, vai beneficiar dos algoritmos anteriores na medida em que continuam a ser eliminadas todas as leituras que são desnecessárias. Em termos de resultados, os ganhos são pelo menos iguais aqueles apresentados pela versão anterior, mas são impossíveis de calcular sem a recolha de dados do uso dos utilizadores finais.

Alterações na Geração do Código das Entidades

No momento da compilação de uma entidade definida em *Service Studio*, é construída a estrutura na linguagem C# que a representa. Para cada uma dessas estruturas, são criados membros privados que representam os atributos declarados pelo programador no momento da definição da entidade e também um conjunto de métodos e operadores. São criadas as operações de comparação habituais para relacionar instâncias da mesma entidade e alguns métodos para: leitura da base de dados, cópia a partir de outra estrutura, duplicação, serialização e construtores, entre outros.

A abordagem seguida para gerir a informação dos atributos que trocaram de valor, passa por acrescentar um membro de dados *BitArray* em cada *Entity Record*, com o tamanho correspondente ao número de atributos declarados. A semântica desse novo campo é a mais simples possível: um atributo tem a sua posição correspondente no *BitArray* como verdadeira se o seu valor mudou em tempo de execução. Optou-se assim pelo nome *ChangedAttributes*.

Em cada operação de afectação passou a ser necessário verificar se o valor que está a ser aplicado ao campo da entidade é diferente do existente, e em caso afirmativo, alterar o valor dessa posição no *BitArray*.

Actualmente, cada afectação é escrita em ficheiro da forma mais simples possível (4.3):

Listagem 4.3: Exemplo de código gerado de uma afectação

```
1 // Query1.List.Current.Person.Name = "newName"
2 localVars.queryResQuery1_outParamList.CurrentRec.ssENPerson.ssName = "newName";
```

Para solucionar este problema a alteração mais directa implicaria encher o código gerado de verificações, o que tornaria o programa bastante mais complicado de analisar e de maiores dimensões.

Optou-se então por realizar todo esta verificação ao nível da entidade. Definiram-se todos os atributos das entidades como privados e criaram-se propriedades [35] (set/get) em C# que fazem as verificações necessárias, antes de alterar o valor. O código seguinte (4.4) mostra um exemplo duma propriedade em C# para manipular um membro privado *counter*.

Listagem 4.4: Exemplo de propriedade em C#

```
1
2  private int _counter;
3
4  public int counter {
5      get {
6          return _counter;
7      }
8      set {
9          _counter = value;
10     }
11 }
```

As propriedades são membros que permitem aceder e alterar os atributos privados de uma classe ou estrutura de forma flexível, como se de um acesso normal a um campo se tratasse. Estas podem ser usadas como membros públicos, mas na realidade são métodos especiais assessores. A grande vantagem é que permitem redefinir a semântica das operações sem grande esforço e sem necessidade de alterar a geração de código das afectações (sem casos especiais).

Devido à partilha de funções geradoras de código entre estruturas e registos, foi necessário introduzir algumas condições, para que as modificações anteriores só se reflictam nos *EntityRecords*. A interface comum *ISimpleRecord* foi também alterada para permitir a cópia

de *BitArrays* entre registos. Isto implicou que as estruturas gerassem uma excepção caso este método seja invocado, pois não possuem esta nova propriedade. No entanto, esta situação não é preocupante pois nunca acontecerá se o código do programa for bem gerado.

Existem outros cuidados a ter na manipulação do BitArray. Na criação de uma instância de uma entidade, os *bits* têm que ser colocados todos a verdadeiro porque os valores da entidade são todos novos. Nos métodos de cópia e duplicação de entidades, os valores têm que ser copiados tal como estão actualmente, para garantir que não se perde a informação que foi alterada. Por fim, no método para leitura de uma entidade da base de dados, o *BitArray* é colocado com todos os atributos a falso, porque estes passam a representar o estado actual da base de dados.

Relativamente à troca de atributos chave, foram tomadas exactamente as mesmas decisões em relação à solução anterior, ou seja, a troca destes valores resulta na alteração de todos os *bits* para verdadeiro.

A listagem 4.5 (na página seguinte) mostra um exemplo dum excerto de código gerado para a entidade *User*, onde se pode observar a verificação de mudança do valor dos atributos e a mudança no *BitArray*.

Alterações no Viewstate

Nas aplicações Web, o contexto de uma página tem que ser enviado para o lado do utilizador (navegador Web), e persistir neste entre *postbacks* (recarregamentos após submissão de formulários). Em aplicações .Net este conjunto de dados que é guardado no navegador é vulgarmente denominado *Viewstate*.

O algoritmo de *Liveness* do compilador também optimiza a informação que necessita de ser enviada para o *Viewstate* com o objectivo de reduzir o seu tamanho, tornando a navegação Web mais rápida. Este contexto é passado para a página durante o carregamento e retido nesta durante os *postbacks*, o que implica que os valores que estão a ser manipulados pelo utilizador precisam de ser constantemente gravados e lidos do *Viewstate*. Desta forma, os *BitArrays* dos *Entity Records* necessitam de acompanhar os atributos das entidades nesta navegação, para que a informação dos atributos modificados nunca se perca.

Um pormenor importante é a necessidade do *BitArray* (*ChangedAttributes*) ser o último campo da estrutura a ser reposto com o valor gravado. Isto é importante porque, durante as leituras dos atributos da entidade para o registo, estes atributos são novamente afectados (porque a afectação usa a propriedade C# em todos os casos), alterando todas as posições do *BitArray* para verdadeiro (*changed*). Se o *BitArray* for carregado no final, os valores originais são recuperados.

Listagem 4.5: Exemplo de código gerado os atributos

```
1
2  public struct ENUserEntityRecord: ISerializable, ISimpleRecord {
3  // ...
4
5  public BitArray ChangedAttributes;
6
7  private int _Tenant_Id;
8
9  public int Tenant_Id {
10    get {
11       return _Tenant_Id;
12    }
13    set {
14       if(_Tenant_Id != value) {
15          ChangedAttributes = new BitArray(11, true); // key modification
16          _Tenant_Id = value;
17       }
18    }
19   }
20
21 private int _ssId;
22
23 public int ssId {
24    get {
25       return _ssId;
26    }
27    set {
28       if(_ssId != value) {
29          ChangedAttributes = new BitArray(11, true); // key modification
30          _ssId = value;
31       }
32    }
33 }
34
35 private int _ssName;
36
37 public int ssName {
38    get {
39       return _ssName;
40    }
41    set {
42       if(_ssName != value) {
43          ChangedAttributes[2] = true;
44          _ssName = value;
45       }
46    }
47 }
```

```
48 // ... (more)
```

Complicações Resultantes da Optimização de Atributos

Aparentemente, a informação adicionada anteriormente seria suficiente para solucionar o problema das optimizações em tempo de execução. Infelizmente as optimizações resultantes do algoritmo de *Liveness* podem causar situações indesejáveis. Vejamos o exemplo da Figura 4.7.

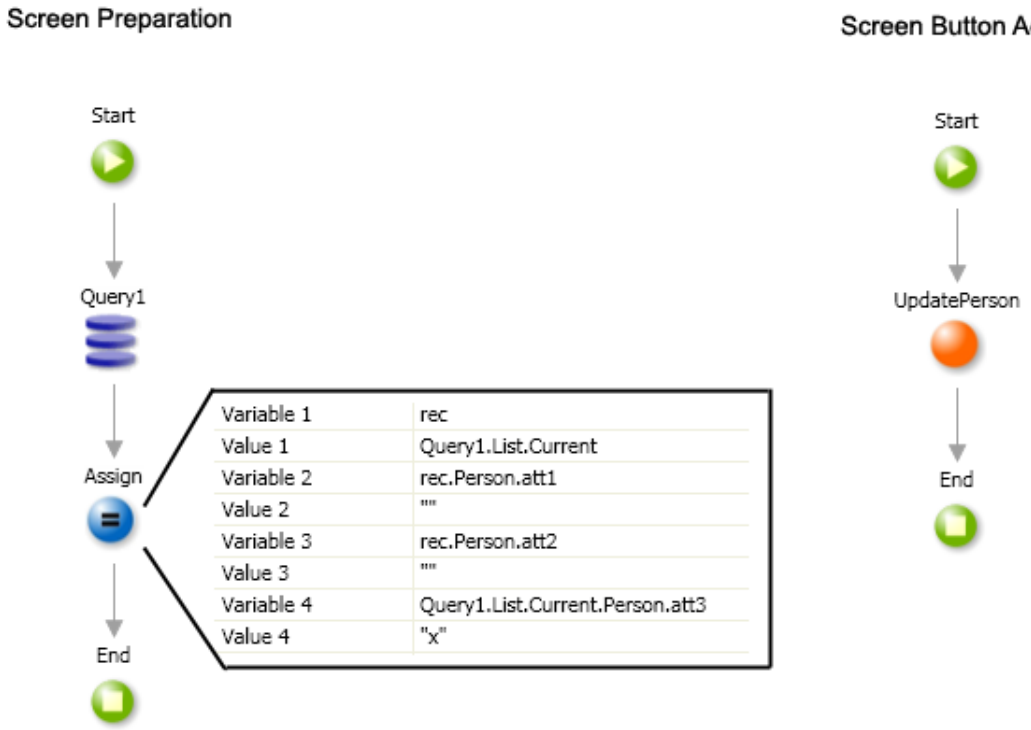

Figura 4.6: Exemplo de *Screen Preparation* e *Action Flow* de uma acção

O *action flow* do lado esquerdo representa a acção de preparação de um ecrã e o do lado direito uma acção de um botão desse mesmo ecrã.

Na preparação do ecrã, a *Query1* retorna uma entidade do tipo *Person* com três atributos do tipo *string* (att1, att2 e att3). De seguida são realizadas quatro operações de afectação. Na primeira, o primeiro registo do resultado da *query* é copiado para uma variável (rec) do mesmo tipo. Depois são afectados os dois primeiros campos da variável com *strings* vazias e o terceiro elemento do *current record* da *Query1* com o valor "x".

Centrando a atenção nas afectações sobre os atributos *att1* e *att2*, reparamos que estes acabam por ser optimizados, porque o seu valor não precisa de ser lido (são substituídos por *strings*

vazias).

Esta sequência permite que a *Query1* apenas traga da base de dados os atributos *Id* e *att3* (este último por causa da limitação do *current record*), o que leva a que no *Entity Record* os dois primeiros atributos sejam preenchidos com os valores por omissão.

O problema deste exemplo é que os valores por omissão das variáveis do tipo *string* correspondem ao valor que está a ser atribuído nas operações, ou seja, a *string* vazia. Assim, mesmo que na base de dados esteja um valor diferente, o *bit* correspondente a cada um destes atributos não vai ser actualizado na acção do botão, porque não foi detectada uma mudança de valor.

Para solucionar este caso especial seguiu-se uma abordagem próxima daquela que foi utilizada para guardar os atributos alterados. A solução passou por inserir mais um *BitArray* nos registos das entidades que indica os atributos que foram optimizados. O objectivo deste novo membro de dados é alterar a semântica da operação *set* da propriedade C# de cada atributo, de forma a considerar que numa afectação, se o valor não mudar mas tiver sido optimizado, então deve ser marcado como alterado para que seja actualizado na base de dados. A listagem 4.6 (na página seguinte) mostra o resultado final em termos de código gerado para cada atributo.

A manutenção do *BitArray OptimizedAttributes* nas entidades não é trivial. Quando são realizadas *queries*, os valores dos atributos optimizados têm que ser propagados desde a lista até aos *Entity Records*. No construtor do *Entity Record* o *BitArray* é inicializado com todos os valores como falsos, ou seja, como não sendo optimizados. Em métodos de duplicação e preenchimento através de outro registo, os valores são copiados, tal como acontecia para o *ChangedAttributes*. Já nó método de leitura de valores da base de dados, assume-se os valores vão ser definidos de seguida pela *query*, não sendo necessário fazer nada. Mas, o processo complica-se mais quando é necessário copiar registos e estruturas, como vai ser descrito na secção seguinte.

Listagem 4.6: Exemplo de código gerado para cada atributo

```
1
2  public struct ENUserEntityRecord: ISerializable, ISimpleRecord {
3  // ...
4  public BitArray ChangedAttributes;
5  public BitArray OptimizedAttributes;
6
7  private int _Tenant_Id;
8
9  public int Tenant_Id {
10    get {
11       return _Tenant_Id;
12    }
13    set {
14       if(_Tenant_Id != value || OptimizedAttributes[0]) {
15          ChangedAttributes = new BitArray(11, true);
16          _Tenant_Id = value;
17       }
18    }
19   }
20
21 private int _ssId;
22
23 public int ssId {
24    get {
25       return _ssId;
26    }
27    set {
28       if(_ssId != value) { // The entity key can't be optimized
29          ChangedAttributes = new BitArray(11, true);
30          _ssId = value;
31       }
32    }
33 }
34
35 private int _ssName;
36
37 public int ssName {
38    get {
39       return _ssName;
40    }
41    set {
42       if(_ssName != value || OptimizedAttributes[2]) {
43          ChangedAttributes[2] = true;
44          _ssName = value;
45       }
46    }
47 }
```

```
48 // ... (more)
```

Propagação dos Atributos Optimizados

No momento da criação de uma entidade em *Service Studio*, são gerados automaticamente métodos para criar, obter, alterar e apagar essa nova entidade na base de dados, de onde se destaca naturalmente a acção *Update<Entity>*. São gerados dois métodos que podem obter entidades com base num identificador: *Get<Entity>* e *GetForUpdate<Entity>*. Estes dois últimos, como não são optimizados pelo algoritmo de *Liveness*, quando capturam uma entidade da base de dados têm que colocar o *BitArray OptimizedAttributes* com todos os valores a falso. O mesmo se aplica às *Advanced Queries* que também não são optimizadas. Neste caso, e à semelhança das *Simple Queries*, é preciso propagar os valores desde a *Record List* até aos *Entity Records*.

A definição de uma entidade em *Service Studio* gera, para além do código do respectivo *Entity Record*, código para o *RecordType* que engloba o *Entity Record* e para uma *Record List* do *RecordType* gerado (ver tipos de dados da linguagem *OutSytems* em 3.3.3).

Quando uma *query* é executada e recolhe dados para o *Current Record*, na realidade executa a operação de leitura da sua *Record List*. Esta operação preenche o *Record Type* do elemento corrente com a informação da base de dados, que depois propaga essa informação para o respectivo *Entity Record*, ou para os vários, no caso de se tratar de uma operação de junção. Da mesma forma que esta informação é passada entre os diversos níveis, foi criado um mecanismo de propagação do *BitArray* dos atributos optimizados.

No momento da definição de uma *Simple Query*, é gerada a informação dos atributos optimizados. Depois da criação da respectiva *Record List* que vai guardar os resultados da *query*, esse *BitArray* é guardado no seu interior. Seguidamente, a operação de leitura da lista encarrega-se de propagar esta informação até ao seu *Record Type* e este até aos *Entity Records*.

Qualquer outro tipo de acção que utilize a operação de leitura (p.e. o nó *ExcelToRecordList*) terá que propagar *BitArrays* com todos os campos a falso, porque apenas as *Simple Queries* são optimizadas.

Outro aspecto importante acontece na cópia de registos (*Record Types*). Na cópia directa de registos (de entidades ou de junções de entidades) é criado dinamicamente um *array* de *BitArrays* para conter a informação de cada uma das entidades interiores. Essa informação é posteriormente passada para o registo afectado. No caso em que os registos são estruturas, os campos que não sejam *Record Types* são inseridos com *null* nessa posição do *array* de *BitArrays* e a cópia é realizada da mesma forma, sendo os valores nulos ignorados no destino.

Toda esta informação tem que ser passada dinamicamente em tempo de execução, porque apesar da informação ser estática, os tipos são partilhados em todo o código, e em circunstâncias diferentes, os campos optimizados podem ser outros.

Optimização do ViewState

À semelhança do que foi referido para os *ChangedAttributes*, este novo *BitArray* também necessita de ser colocado no *Viewstate*. Contudo, é precisamente neste ponto que surge outro pormenor importante. O algoritmo de *Liveness* tem como objectivo, para além de optimizar os valores lidos das *Simple Queries*, optimizar os valores que são enviados para o *Viewstate*. Isto implica que quando as entidades forem lidas do *Viewstate* possam ter mais atributos optimizados do que tinham no momento em que lá foram guardadas. A solução passa por guardar no *Viewstate*, no momento da gravação do *BitArray*, o resultado da operação lógica *OR* entre os *OptimizedAttributes* da entidade e os do *Viewstate*. Assim, quando a entidade for reposta, já vai ter a informação correcta de todos os campos optimizados em ambas as situações.

Com esta última alteração ficou completa a implementação deste trabalho, desta vez com actualizações apenas dos valores alterados em tempo de execução. A secção seguinte descreve o processo de validação da solução implementada.

4.2.3 Validação

Esta secção do documento pretende clarificar o trabalho feito para validação da solução implementada e está repartida em duas subsecções. A primeira revela os métodos para validação da correcção da solução ao nível da forma em como foram realizados os testes e alguns detalhes sobre os mesmos. A segunda parte baseia-se na amostragem e análise crítica dos resultados obtidos nos testes de performance realizados após a fase de implementação.

4.2.3.1 Correcção

O trabalho realizado em termos de validação de correcção assenta na existência de um mecanismo, interno à *OutSystems*, com capacidade de correr todos os dias, cerca de 3500 testes de regressão [39] (testes que verificam se as alterações do dia anterior estragaram os resultados dos testes). Esta optimização implementada, teve impacto em alguns testes anteriores, que tiveram que ser analisados e corrigidos à medida que todo este código se foi juntando com o restante código em desenvolvimento. Para cada uma das iterações deste projecto, foram criados testes específicos para validar a sua correcção. Podem-se distinguir três tipos de testes: testes em tempo de compilação, testes de navegador simples e testes de navegador com captura informação de tempo de execução.

Para a realização de testes em tempo de compilação, o mecanismo de asserções da linguagem *OutSystems* foi estendido para verificação dos atributos utilizados em cada operação. Esta verificação passa por comparar os usos da operação *Update<Entity>*, gerados em tempo de compilação, com aqueles que são declarados na *string* de comentário. Estas asserções são bastante similares com as existentes para as *Simple Queries* onde são declarados os campos que

são obtidos da base de dados (não optimizados pelo algoritmo de *Liveness*). As asserções que não coincidirem com a informação estática, têm a capacidade de forçar a que a operação seja realizada com a informação declarada.

No exemplo da figura 4.7 são mostrados dois exemplos de asserções incoerentes com as optimizações, que resultam em mensagens de aviso para operações forçadas (Figura: 4.8). É atribuído um novo valor à idade e ao nome da pessoa em questão, mas como na asserção são indicados para actualização os atributos *name* e *username*, são esses que vão ser actualizados.

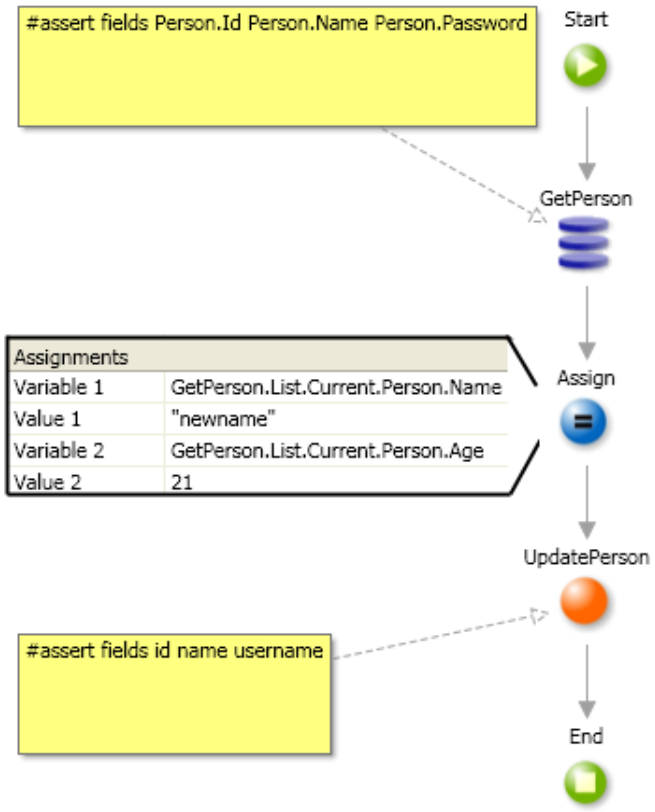

Figura 4.7: Exemplo de asserções em código *OutSystems*

Estas asserções foram testadas individualmente com casos muito simples para verificar a sua correcção. Em termos de implementação, estão integradas no algoritmo 4.2.2 no momento de corte dos usos. Os valores no comentário são comparados com os valores de uso no nó, gerando as mensagens de aviso e forçando novos usos.

Com a possibilidade de gerar estas asserções foram criados vários testes que englobam

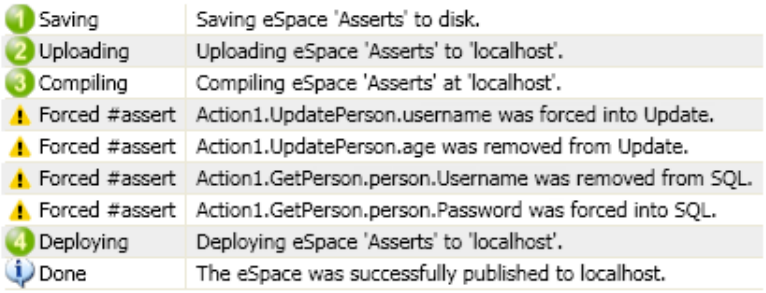

Figura 4.8: Mensagens de aviso em asserções para o exemplo da figura 4.7

todas as situações relevantes de testar em tempo e compilação desde actualizações em ciclos e decisões até passagem de atributos modificados para acções externas e internas ao *eSpace*.

Para testar as optimizações tem tempo de execução, estas verificações com base em asserções não são suficientes. Foram produzidos testes de navegação em browser, com base em *scripts Selenium IDE* [40] com asserções ao nível da informação mostrada ao utilizador no navegador. Estes testes permitem simular navegação entre páginas e preenchimento de formulários. Para além de estarem mais próximos do comportamento dos utilizadores finais, permitem verificar através do carregamento das páginas e da informação mostrada ao utilizador, se as alterações são reflectidas na entidade da forma correcta. Permitem ainda simular o comportamento de utilizadores no preenchimento de formulários concorrentes e a correcção dos resultados. A geração destes *scripts* é feita gravando a navegação de um utilizador numa página *Web* normal, permitindo a cada momento fazer asserções de texto. Se a navegação ou as asserções falharem, o teste falha por completo.

Os testes em *Selenium IDE* são bastante completos, mas não permitem visualizar o estado das entidades manipuladas. Para conseguir tal informação, foram geradas acções especiais através do *Integration Studio* que recebem *Entity Records* e analisam o estado dos vectores de *bits*. Esta informação é capturada através de *reflection* de objectos em *C#*. Esta habilidade permite retirar informação dinâmica sobre os objectos (métodos e atributos) em tempo de execução. O uso destas acções especiais permitiram a criação de testes onde o resultado da validação do estado dos vectores de *bits*, foi mostrado em tempo de execução e depois alvo de asserções.

A combinação desta informação com as potencialidades dos testes de navegador geram informação suficiente para a verificação completa da validade da implementação.

É importante salientar que a criação destes testes foi acompanhada por especialistas na linguagem para garantir que abrangem todas as situações, incluindo as mais invulgares.

4.2.3.2 Avaliação dos Resultados

A segunda parte desta secção de validação visa analisar os resultados obtidos com os testes de performance. Para o efeito foram seleccionados todos os *eSpaces* de uma solução de grandes dimensões juntamente com alguns outros *eSpaces* relevantes (p.e. *Service Center*).

Na altura em que estes resultados foram obtidos, nem todos os ficheiros passavam pelo actualizador de versão, por isso alguns tiveram que ser deixados de parte. Os testes basearam-se na recolha de dados da compilação sucessiva de 47 ficheiros *OML*, onde em cada um deles foi compilado 4 vezes sem as optimizações e 4 vezes com as optimizações, assumindo o melhor caso: todas as entidades com a propriedade *Update Behavior* marcada como *Changed Attributes*.

Todos os gráficos apresentados nesta secção representam um subconjunto de todos os dados recolhidos nesta fase do projecto. Como informação complementar a este documento, podem ser encontrados em anexo os detalhes da compilação de cada *eSpace*, incluindo valores minimos, máximos, médias e desvios padrão. Em anexo encontram-se também gráficos semelhantes aos apresentados nesta secção, mas com informação percentual.

Estes testes só mostram os ganhos nas aplicações obtidos em tempo de compilação. As optimizações em tempo de execução terão sempre, na pior das hipoteses, o conjunto dos valores a enviar para as operações de *Update*, igual aqueles que seriam enviados em tempo de compilação. Isto porque, nesta última, assume-se que todas as atribuições de valores a variáveis resultam em troca de valor. Para calcular ganhos aproximados em tempo de execução dos usos dos *Updates*, seria necessário recolher métricas de utilização das aplicações, e depois multiplicar a percentagem de afectações que resultam em troca de valor pelos resultados finais.

Ganho Global

As alterações introduzidas por este trabalho afectam directamente os usos das operações de *Update<Entity>*, que por sua vez influenciam as definições de valores realizadas nas *Simple Queries*. Em termos gerais o resultado pode ser observado de forma pouco detalhada pelo gráfico da figura 4.9. A análise indica que, em tempo de compilação, mais de 50% dos valores envolvidos em operações de *Update* nunca serão alterados, e por isso são optimizados. As *Simple Queries* ganham numa redução superior a 10% em definição de valores, ou seja, menos atributos trazidos da base de dados.

A informação anterior é muito pouco precisa porque não têm em conta os tipos de dados envolvidos. As operações são mais lentas, quanto maior for a soma do tamanho de todos os atributos envolvidos da entidade enviados. Estes atributos podem pertencer a qualquer um dos tipos: *String*, *Integer*, *Decimal*, *Boolean*, *DateTime* e *Byte*[].

Os valores dos tipos *String* e *Byte*[] têm dimensão variável na linguagem e não é possível definir um limite para este último. Em termos de relevância, a ordem é genericamente a seguinte: *Byte*[] (dimensão variável), *String* (número de caracteres * 16 *bit*), *Decimal* (128 *bit*), *DateTime* (64 *bit*), *Integer* (32 *bit*) e *Boolean* (8 *bit*). Para efeitos de avaliação de resultados, o tipo *String* foi agrupado em 4 intervalos (por número de caracteres): [0, 49], [50, 199], [200, 799] e [800, max].

Todos os testes realizados e mostrados de seguida têm em conta esta separação de tipos.

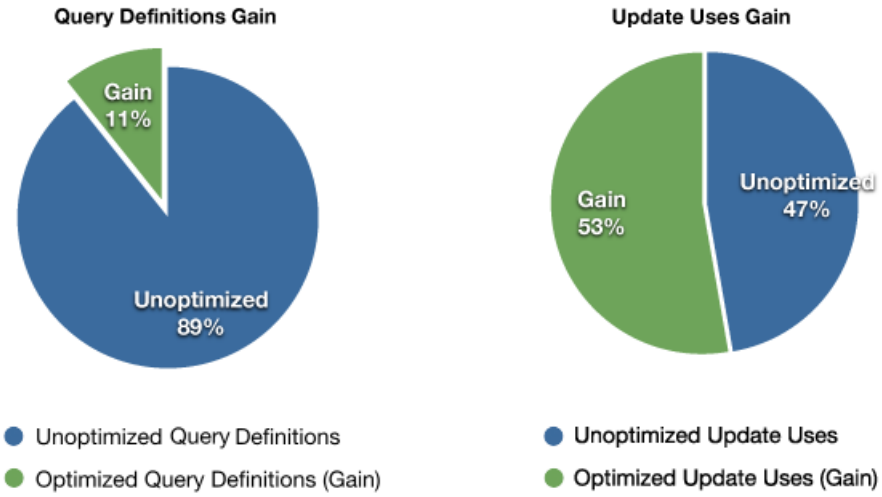

Figura 4.9: Ganhos gerais com análise estática de código

Ganho em Operações de Update

Observando em mais detalhe os *Updates*, pode-se verificar pelo gráfico da figura 4.10, que os tipos mais manipulados são as *strings* e os números inteiros. A redução de valores actualizados ronda, em todos os casos (excepto nos números decimais), os 40% e 55% da quantidade original. Em termos de performance, e tendo em conta o tamanho dos tipos de dados, as maiores influências notam-se na forte redução nos tipos *string*. Apesar de o impacto de performance sentir-se mais em actualizações de atributos binários (*byte*[]), a sua presença mostra-se rara nas aplicações. Ainda assim, um potencial de ganho de 40% é bastante significativo.

Estes resultados em elementos do tipo *Byte*[], devem-se também à existência de uma mensagem de aviso que alerta os programadores para inserir elementos de tipo binário em entidades separadas. Isto leva a que as actualizações a entidades com estes atributos sejam quase sempre optimizadas, no sentido em que não existem muitas situações em que há actualizações exclusivas a outros atributos quando as entidades possuem dados binários.

Pela análise do gráfico da figura 4.11, verificamos que a percentagem de ganho não varia muito consoante a dimensão das *strings* envolvidas. A maior parte destes atributos ronda valores entre 50 e 200 caracteres, mas existe um elevado número de casos acima destes valores.

Para efeitos meramente ilustrativos, vamos ignorar a presença de atributos binários (muito pouco frequentes e de tamanho muito variável) e assumir que é preenchido 1/3 do valor máximo de cada um destes grupos de *strings*, tendo uma média de 800 caracteres para o último conjunto. No agregado de todos os testes foram encontradas 248 operações de *Update*, que tendo em conta o tamanho (indicativo) de cada tipo de dados e os valores obtidos nos testes,

cada operação submeteria anteriormente um valor próximo de 1,98 KB. Com optimizações e com as mesmas aproximações este valor desce para os 1,00 KB, ficando muito próximo daquele estimado apenas pela redução do número de usos, estando 4% apenas abaixo do ganho esperado.

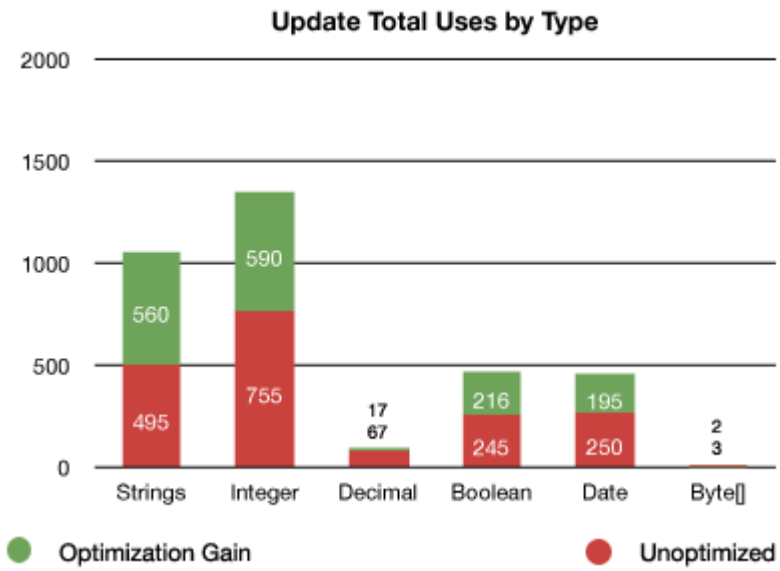

Figura 4.10: Ganho em usos por tipo em *Updates* nos testes realizados

Ganho em Simple Queries

As *Simple Queries*, na versão anterior da plataforma, já estavam bastante optimizadas no que diz respeito à recolha de valores das bases de dados. No entanto, supunha-se que ao retirar usos das operações de *Update*, estas pudessem descartar alguns valores que anteriormente eram lidos e assim aumentar a eficiência global das aplicações. Os resultados mostraram-se bastante surpreendentes. Com excepção dos tipos *Byte*[] todos os ganhos foram pelo menos iguais a 10%.

Dentro dos elementos de tipo *String*, o ganho foi ligeiramente mais significativo naquelas de maior dimensão talvez por ser menos usual a edição de textos grandes em formulários.

Nos atributos binários a variação foi muito pequena, fazendo que os 2 elementos que foram optimizados apenas reflectissem 3% do número total, uma vez que já existe uma forte optimização deste tipo de dados.

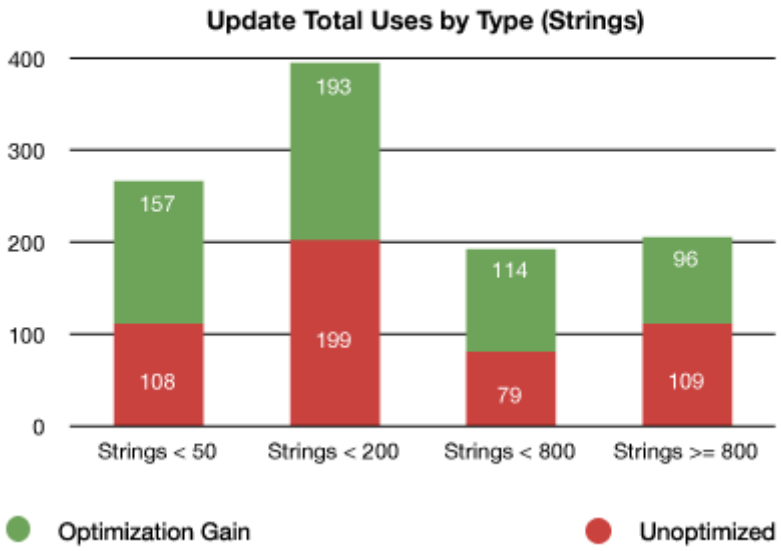

Figura 4.11: Ganho total de usos por tamanhos de *strings* em *Updates* nos testes realizados

À semelhança do que foi feito para o exemplo anterior, como exemplo ilustrativo nas mesmas condições e tendo em conta que em todos os testes havia um total de 2496 *Simple Queries*, anteriormente cada uma destas operações recolhia um conjunto de dados perto dos 0,55KB. Com a nova versão esse valor desce para os 0,48KB, ficando 2% acima do valor esperado pelos cálculos iniciais, o que é perfeitamente natural tendo em conta as aproximações e a variação esperada pelas diferenças dos tipos de dados.

Modificações no ViewState

As repercussões nos valores que precisam de ser enviados para o *viewstate* são mostradas pelo gráfico 4.14. Existe uma redução de 5% nos tipos *String, Integer, Boolean* e *Date*. Embora este valor não seja muito elevado, é importante que seja suficientemente significativo para compensar a introdução de variáveis do tipo *BitArray*. A classe *BitArray* é implementada em C# com números inteiros, o que implica que numa entidade até 32 atributos esta estrutura terá apenas um número inteiro. Tipicamente este valor nunca será ultrapassado, mas tendo em conta os dados obtidos nos testes, os ganhos anteriores terão idealmente que compensar a introdução de 2712 novos *BitArrays* no *Viewstate*.

Infelizmente não existe nenhuma forma fácil de calcular o tamanho do *Viewstate* nas aplicações. Até mesmo aproximações à semelhança das anteriores, acabam por estar demasiado

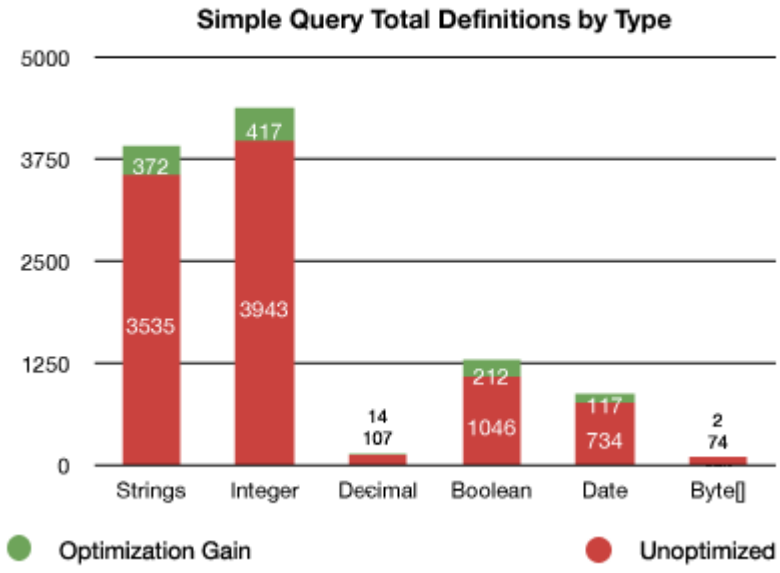

Figura 4.12: Ganho em definições por tipo em *queries* nos testes realizados

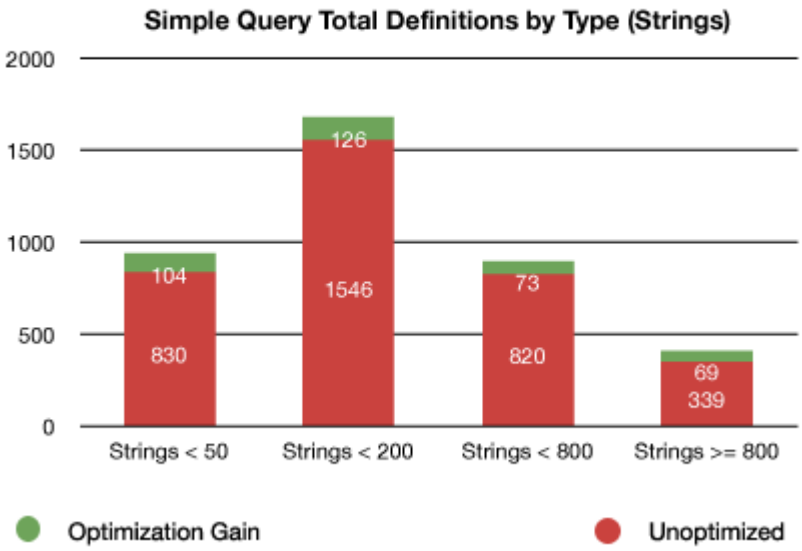

Figura 4.13: Ganho total de definições por tamanhos de *strings* em *queries* nos testes realizados

distantes dos valores reais, porque o conjunto de valores é alvo de um algoritmo de compressão.

Mas, assumindo uma compressão com ganhos uniformes e as condições anteriores de aproximação aos tamanhos dos tipos de dados, são poupados no *Viewstate*, em relação à versão antiga, (sem contar com os elementos *BitArray*) 74,21KB. Se assumirmos, por excesso, que cada *BitArray* ocupa 128 *bits* (equivalente a 4 números inteiros), o conjunto de 2712 valores deste tipo ocupará 42,38KB, o que significa uma redução media positiva. Isto indica que, em média, existe um ganho potencial de 3% no *Viewstate* das aplicações, antes de ser aplicado o algoritmo de compressão, o que implica que o *Viewstate* também beneficiará com o conjunto de optimizações realizadas. Isto deve-se ao facto de uma *string* de 800 caracteres (por exemplo), com estas medidas de aproximação, ser equivalente a 100 elementos *BitArray*.

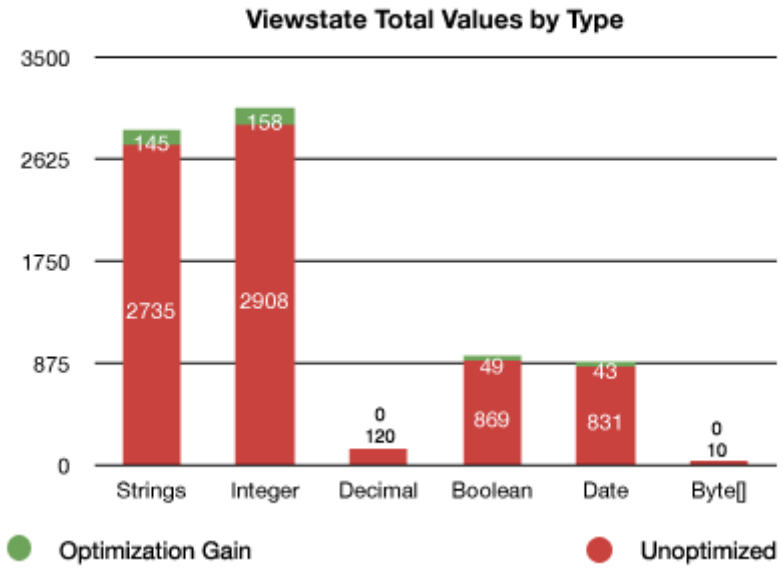

Figura 4.14: Ganho no envio de valores para *Viewstate* por tipo nos testes realizados

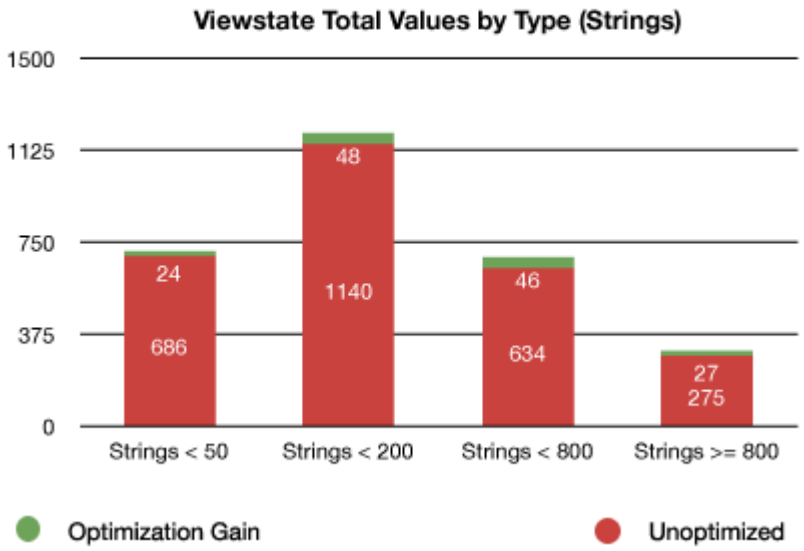

Figura 4.15: Ganho total por tamanhos de *strings* enviadas para o *Viewstate* nos testes realizados

Impacto no Compilador

A integração dos algoritmos implementados no compilador teve um impacto bastante reduzido, no que diz respeito a variação no tempo de compilação em utilização de memória. Os testes mostraram que nos piores casos o tempo de execução dos dois algoritmos chega a valores próximos de 2% do tempo total de compilação, e na grande maioria dos testes não ultrapassa 1% desse valor (Figura: 4.16).

Já em relação a utilização de memória os valores em 4.17 mostram que a variação é rara e inconstante. Os valores variam quase sempre abaixo do desvio padrão (ver tabela B.3), o que não permite obter resultados conclusivos. O comportamento inconstante do *garbage collector* do C# está na base desta oscilação.

Como conclusão, podemos salientar que as perdas em tempo de execução e as variações não deterministas na utilização de memória, não são suficientes para invalidar os ganhos anteriores. Os ganhos são bastante expressivos, tanto nas operações *Update<Entity>* como no conjunto das *Simple Queries*, embora nestas já se esperasse que fosse significativamente mais fraco. A ameaça de crescimento do *Viewstate* com os novos tipos *BitArray* é também compensada, e até genericamente ultrapassada, pelos efeitos laterais das optimizações anteriores em relação a definições e usos.

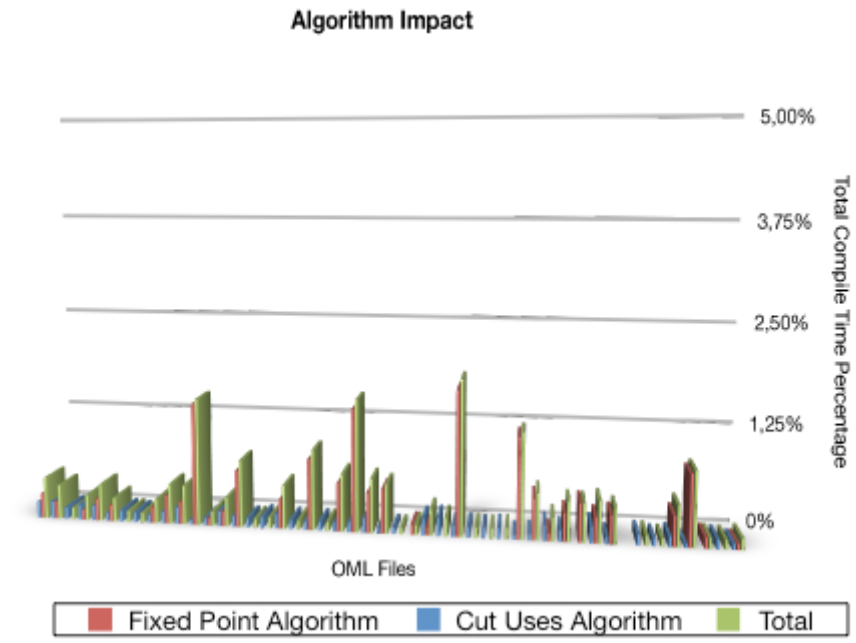

Figura 4.16: Percentagem do tempo de compilação gasto pelos novos algoritmos

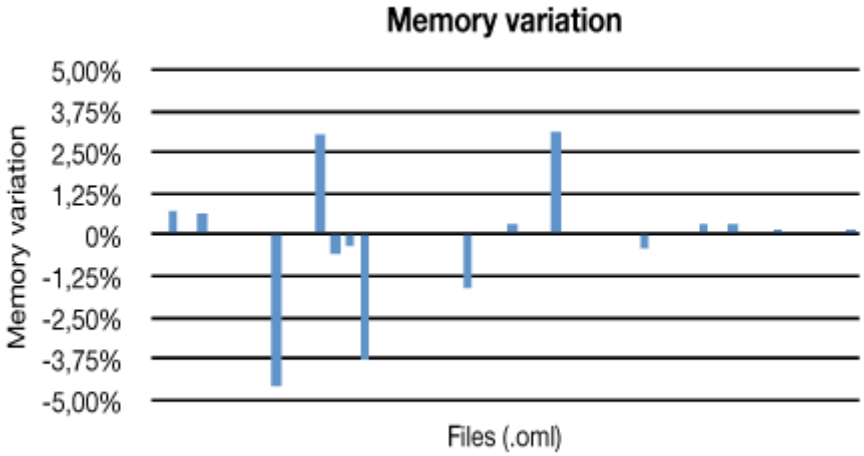

Figura 4.17: Variação de memória nos testes realizados em relação à versão sem optimizações

4.3 Trabalho Experimental

Este trabalho foi realizado em ambiente empresarial nas instalações da equipa R&D (*Research & Development*) da *OutSystems*. A fase de implementação da solução foi realizada em completa integração com a equipa responsável pelo desenvolvimento da versão 5.0 da *Agile Platform* seguindo metodologias ágeis de desenvolvimento de *Software* [41].

Numa fase inicial foi necessário compreender as funcionalidades e utilização dos diversos componentes da *Agile Platform*: *Service Center*, *Integration Studio* e *Service Studio*. Neste último foi fundamental perceber a linguagem a optimizar e as necessidades actuais, no que diz respeito a cenários de possível optimização e aos seus pós e contras. Depois do levantamento destes padrões, foram definidas prioridades e definido o rumo do projecto no sentido de optimizar apenas as operações de *Update<Entity>*.

A fase inicial do projecto foi essencialmente de pesquisa e compreensão do código existente. A *Agile Platform* é constituída por mais de 70 projectos em *Microsoft Visual Studio 2008* numa base que supera 1 milhão de linhas de código. Foi importante focalizar o processo nas parcelas de código relevantes, nomeadamente no compilador, optimizador e geração de código.

O processo de implementação foi realizado em código sob constantes alterações provenientes da equipa de desenvolvimento, o que levou a trabalhar muitas vezes com versões instáveis da plataforma. Esta instabilidade foi responsável por corromper ficheiros de teste (*.oml*), o que implicou, a certa altura, a ter sempre uma versão estável (versão 4.2) e proceder ao respectivo *upgrade* quando necessário. Em relação ao desenvolvimento em si, foi um processo iterativo marcado por algumas mudanças de opinião e indecisões, muitas delas em prol da usabilidade, mas também devido às delicadas decisões envolvendo alterações semânticas. O aparecimento de novos cenários de utilização que punham em causa a solução iam levando ao reforço a solução e a repensar se o caminho seguido tinha sido o mais correcto. Por isso, foi importantíssimo ter por perto especialistas na linguagem para pensar nestes cenários de utilização e indicar possíveis falhas.

A criação de testes foi constante ao longo do desenvolvimento. Estes testes foram colocados numa máquina dedicada (*Regression Tests Machine*) que corre todos os dias, todos os testes de regressão existentes (cerca de 3500), de forma a mostrar como é que as alterações do dia anterior os afectaram. Isto levou a ter que perceber e aceitar (ou não) as diferenças no *output* de alguns deles, provocadas pelas alterações introduzidas.

Por fim, todo este esforço foi recompensado com a integração total das optimizações na base de código da versão 5.0 do *Service Studio*, com data de lançamento marcada para o final de 2009.

5

Conclusão

As linguagens de domínio específico, caracterizadas por elevados níveis de abstracção, apresentam geralmente desafios muito interessantes para optimizações. O código gerado pelo compilador muitas vezes não é optimizado e introduz *overheads* consideráveis nas aplicações finais, resultantes de redundâncias, má estruturação ou excesso de código.

Quando estas linguagens combinam lógica aplicacional com consultas e comandos sobre bases de dados, surgem situações onde a ordenação, antecipação, união ou pré-cálculo das *queries* pode ter impacto significativo nas aplicações. Por outro lado, a análise do fluxo de dados permite também reduzir o fluxo de comunicação entre os servidores e as aplicações aumentando assim a sua eficiência.

Neste contexto apresentámos e estudámos a linguagem *OutSystems*, identificando seis padrões de código sub-optimizados comuns nas aplicações. Para cada um destes padrões, apresentámos uma solução geral de optimização, assim como a análise das possíveis alterações semânticas. Devido ás limitações temporais, foi apenas possível implementar a optimização mais relevante relativa à actualização parcial de entidades na linguagem *OutSystems*.

Através da introdução de um algoritmo de análise de fluxo de dados no compilador, tornou-se possível propagar escritas sobre os atributos de entidades até às respectivas operações de actualização, eliminando 53% da informação que era enviada anteriormente nestas operações. Ainda em tempo de compilação, a informação dos atributos desnecessários passou a ser propagada, por análise *liveness*, para as *queries* que os recolhem das bases de dados, introduzindo um ganho de 11% nos dados extraídos em consultas simples. Com esta optimização reduziu-se ainda o tamanho do *Viewstate* das páginas Web em cerca de 3%.

Por cima destas optimizações, foi criado um mecanismo para verificação, em tempo de

execução, dos atributos que trocaram de valor nas afectações. Esta optimização permitiu reduzir ainda mais a informação enviada para as actualizações de entidades, embora o seu ganho potencial seja difícil de calcular sem métricas do comportamento dos utilizadores.

A correcção da solução foi validada através da implementação de um mecanismo de asserções que, em tempo de compilação, verifica se os atributos optimizados pelo algoritmo são os esperados. Os testes de asserções, compilação e de browser foram colocados em maquinas de testes de regressão para garantir a estabilidade desta solução.

Todos os resultados experimentais obtidos resultam da compilação sucessiva de quase 50 aplicações reais de forma a garantir que o impacto final calculado está o mais próximo possível da realidade.

Finalmente, todas as modificações realizadas no compilador no contexto da introdução de actualizações parciais de entidades, foram integradas com sucesso no código do compilador da versão 5.0 da plataforma *OutSystems*.

5.1 Trabalho Futuro

O trabalho realizado no âmbito desta dissertação de mestrado poderá progredir em duas vertentes. A primeira passará por completar o trabalho realizado na implementação da optimização das actualizações de entidades, principalmente na vertente de detecção de conflitos nas escritas na base de dados. O segunda aspecto passará num estudo mais aprofundado e implementação dos restantes padrões analisados no início deste trabalho, seguindo a mesma metodologia: alinhando os objectivos de performance às necessidades reais dos programadores.

Em relação ao trabalho desenvolvido há alguns aspectos que podem ser melhorados. Em primeiro lugar surge a necessidade de incluir um mecanismo de detecção de conflitos nas escritas na base de dados. As entidades deverão possuir uma nova propriedade para suportar detecção de conflitos. Em termos de implementação, seleccionar esta nova propriedade poderá significar acrescentar um novo atributo na entidade que representará a data da última escrita na base de dados. Assim, no momento da escrita, garante-se que os novos valores são escritos apenas se a data da última alteração, que foi lida, corresponder àquela que está na base de dados no momento da actualização. Esta actualização deve ainda incluir a mudança deste valor com a data corrente. Isto permite que a operação seja executada num único passo, sem bloqueio dos dados, da seguinte forma:

$$UPDATE\ Entity\ SET\ LastModified = myCurrentDate, \ldots.$$
$$WHERE\ Id = myId\ AND\ LastModified = myLastModified;$$

O problema desta solução é que não se consegue distinguir se a actualização não foi realizada por existir um conflito na escrita ou por desaparecimento da entidade com aquele identificador. Nesta situação, é preferível manter a operação eficiente e realizar uma segunda *query*, apenas quando a excepção for capturada, para verificar se a entidade foi removida. Desta forma,

só são realizados dois acessos à base de dados se existir um conflito ou se a entidade for removida.

Se fosse decidido que seria necessário distinguir em que atributo(s) existiu o conflito, poderia-se optar por guardar para cada deles alguma informação relativa ao valor lido originalmente da base de dados. Neste caso, a duplicação de cada atributo, mesmo que seja feita apenas nas entidades com esta opção seleccionada, terá efeitos muito negativos no tamanho *Viewstate* das páginas. Uma adaptação da solução anterior solucionaria o problema, onde cada atributo teria informação sobre a última modificação, em forma de data ou contador (que ocupa menos espaço). A lógica a seguir seria exactamente a mesma, apenas com mais verificações.

Ainda no contexto da optimização realizada, existem pontos onde os efeitos a propagação de valores marcados como *changed* poderiam ser mais notados. As operações de *Get<Entity>* não são optimizadas pelo compilador, pelo que possuem código genérico que recolhe sempre todos os atributos da entidade da base de dados. O objectivo seria construir a *query SQL* com informação do algoritmo de *liveness* de forma a capturar apenas os campos necessários. A informação das variáveis que estão activas após a operação de *Get<Entity>* seria passada sob a forma de um *BitArray* à semelhança do que foi feito nos nós *Update<Entity>*.

Outra optimização que iria reforçar os resultados obtidos, seria a resolução (pelo menos parcial) da limitação do compilador em afectações sobre o registo corrente da *Record List* de uma *query*. Actualmente, o algoritmo de *liveness* tem acção limitada ao remover leituras desnecessárias de *queries*. Isto acontece porque assume o pior caso, ou seja, que o registo corrente pode mudar durante a execução, e desta forma as escritas sobre ele não são optimizadas. Um dos principais motivos para que esta optimização não tenha sido realizada até à data, advém da impossibilidade do optimizador olhar para o interior dos *action flows* dos nós *execute action*, que podem até ser referencias a acções de *eSpaces* externos. Nestas situações, o optimizador assume que é necessário que a *query* retorne todos os atributos, porque não tem forma de os calcular correctamente. Esta optimização é decerto um grande desafio mas, até a simples verificação da mudança do *current record* nas restantes situações, poderá ter um impacto significativo.

Como trabalho futuro fica também a oportunidade de reduzir os efeitos da compilação dos diferentes módulos parcialmente. Se for possível obter informação acerca do que acontece dentro de cada acção, os valores de entrada e saída dessa acção poderão também ser optimizados. Com esta alteração, conseguiríamos obter impacto nos resultados apresentados neste trabalho porque em muitos casos é frequente encontrar acções cuja chamada marca todos os valores como lidos.

Anexos

Ganhos em tempo de compilação

A tabela seguinte dispõe os dados recolhidos, em tempo de compilação, do número total de definições realizadas por *Simple Queries* e de usos das operações *Update<Entity>*, em 48 ficheiros OML diferentes. Em ambos os casos, são comparadas as diferenças entre versões e apresentado o respectivo ganho com as alterações (em percentagem).

		Simple Query Definitions			Update Entity Uses		
File (.oml)	**size (Kb)**	**defs(old)**	**defs (new)**	**gain %**	**defs(old)**	**defs(new)**	**gain %**
aad_Lookups	254	79	77	2.53	6	4	33.33
aad_Resources	387	207	203	1.93	31	18	41.94
aad_Traceability	18	0	0	0	0	0	0
aad_training	606	448	440	1.79	56	42	25.00
aad_Training_Exams	207	93	81	12.90	18	3	83.33
AuditEvents	104	39	39	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0
ChartingServices	2655	10	10	0	0	0	0
DealReg	368	168	151	10.12	60	24	60
DealRegBO	219	114	80	29.82	88	34	61.36
ECHO_UI_v1908	3044	1475	1422	3.59	216	94	56.48
eMailServices	299	161	112	30.43	117	51	56.41
EM_Impersonate	120	116	55	52.59	82	20	75.61
EnterpriseManager	1129	986	947	3.96	146	39	73.29
FCK_Editor	669	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0
Licensing	570	257	191	25.68	305	225	26.23
Licensing_SOA	67	0	0	0	0	0	0
Network	747	128	81	36.72	57	8	85.96
NetworkAccessControl	43	3	3	0	0	0	0
NetworkDocuments	555	227	208	8.37	43	15	65.12
NetworkMembers	1372	1177	992	15.72	520	258	50.38
NetworkSolutions	830	391	288	26.34	214	74	65.42
NetworkSupport	275	114	114	0	0	0	0
Network_Emails	270	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0
Partner	472	347	209	39.77	192	54	71.88
PlatformCommon	119	0	0	0	0	0	0
PlatformSegmentation	674	662	659	0.45	29	17	41.38
ProgressBar	20	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0
ServiceCenter	5243	2120	1888	10.94	658	321	51.22
SForce	180	11	11	0	0	0	0
SimpleInbox	105	17	17	0	33	33	0
SizingCatalog	282	317	267	15.77	147	71	51.70
SizingEngine	162	0	0	0	0	0	0
SizingValidation	118	13	13	0	128	128	0
Sizing_Schedule	134	0	0	0	0	0	0
Social	24	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0
TechCenter_Digest	288	52	52	0	10	10	0
Training_Management	1722	841	829	1.43	284	272	4.23
Utility	23	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0
	Sum	10573	9439	10.73	3440	1815	52.76
	Average	224.96	200.83	10.73	73.19	38.62	52.76

Tabela A.1: Listagem das diferenças totais em usos e definições

Impacto no compilador

As tabelas seguintes ilustram o impacto que as optimizações tiveram no compilador da linguagem. Os testes foram realizados sobre 48 ficheiros OML que foram compilados 4 vezes em cada versão da plataforma (com e sem optimizações).

A primeira tabela compara os tempos totais de compilação entre as diferentes versões e apresenta os valores mínimos, máximos, média, desvio padrão e percentagem do desvio padrão sobre o tempo total. Todos os tempos estão em segundos.

A tabela seguinte apresenta os tempos de execução dos dois algoritmos implementados com os mesmos dados da tabela anterior mais a percentagem de cada um sobre o tempo total de compilação.

A última tabela desta secção mostra as diferenças em termos de utilização de memória na compilação das aplicações.

		Compile Time (old version)					Compile Time (new version)					
File (.oml)	**size (Kb)**	**min (s)**	**mean (s)**	**max (s)**	**dev (s)**	**dev %**	**min (s)**	**mean (s)**	**max (s)**	**dev (s)**	**dev %**	**loss %**
aad_Lookups	254	8.89	9.29	9.47	0.27	2.91	8.7	9.73	10.47	0.86	8.84	4.75
aad_Resources	387	13.23	15.09	19.23	2.83	18.75	11.25	14.09	17.02	3.23	22.92	-6.65
aad_Traceability	18	6.42	7.33	8.31	1.01	13.78	5.8	6.9	8.25	1.01	14.64	-5.92
aad_training	606	16.11	17.2	18.02	0.8	4.65	14.92	15.97	16.92	0.82	5.13	-7.18
aad_Training_Exams	207	9.98	11.27	12.67	1.45	12.87	9.92	11.05	12.45	1.25	11.31	-2.01
AuditEvents	104	10.14	10.34	10.53	0.16	1.55	9.91	10.23	10.53	0.26	2.54	-1.06
CalendarWidget	58	4.89	5.02	5.33	0.21	4.18	7.14	7.41	7.92	0.36	4.86	47.74
CaptchaWidget	27	5.3	5.96	7.5	1.03	17.28	7.92	8.43	9.47	0.72	8.54	41.32
ChartingServices	2655	6.42	7.56	9.05	1.27	16.80	9.53	10.22	11.2	0.79	7.73	35.24
DealReg	368	14.25	14.89	15.23	0.44	2.96	12.41	14.2	15.19	1.25	8.80	-4.67
DealRegBO	219	10.73	13.47	16.23	3.04	22.57	13.02	14.49	16.64	1.75	12.08	7.57
ECHO_UI_v1908	3044	60.05	64.12	68.5	4.49	7.00	63.69	68.56	73.06	4.68	6.83	6.93
eMailServices	299	14.34	14.61	15.09	0.33	2.26	10.11	11.61	15.53	2.62	22.57	-20.49
EM_Impersonate	120	10.64	11.56	12.73	1.05	9.08	7.91	11.14	13.58	2.47	22.17	-3.62
EnterpriseManager	1129	27.31	29.34	33.17	2.61	8.90	28.64	31.42	38.67	4.85	15.44	7.11
FCK_Editor	669	10	10.07	10.11	0.05	0.50	7.19	9.35	10.23	1.45	15.51	-7.1
jQueryTooltip	17	7.33	7.39	7.44	0.05	0.68	7.36	7.48	7.58	0.09	1.20	1.11
Licensing	570	15.25	19.78	24.55	4.41	22.30	20.42	22.31	25.05	2.28	10.22	12.8
Licensing_SOA	67	8.16	8.79	10.34	1.04	11.83	8.27	8.87	10.38	1.01	11.39	0.89
Network	747	17.3	19.14	23.69	3.04	15.88	17.61	18.27	19.64	0.93	5.09	-4.57
NetworkAccessControl	43	8	8.36	9.28	0.62	7.42	8.14	9.16	10.48	1.09	11.90	9.62
NetworkDocuments	555	17.94	18.23	18.58	0.33	1.81	17.95	18.48	19.08	0.56	3.03	1.35
NetworkMembers	1372	29.47	32.96	38.23	3.73	11.32	31.36	34.52	42.88	5.58	16.16	4.74
NetworkSolutions	830	20.7	21.16	21.75	0.44	2.08	20.48	22.61	28.09	3.68	16.28	6.89
NetworkSupport	275	13.97	15.61	17.72	1.8	11.53	13.66	14.93	17.48	1.74	11.65	-4.35
Network_Emails	270	12.64	12.83	13.08	0.19	1.48	12.78	13.16	14.03	0.59	4.48	2.59
Paging	26	7.8	8.07	8.69	0.42	5.20	7.66	8.11	8.86	0.54	6.66	0.53
Partner	472	17.05	18.47	20.67	1.58	8.55	17.11	19.55	24.92	3.63	18.57	5.88
PlatformCommon	119	8.83	8.96	9.11	0.14	1.56	8.83	9.04	9.36	0.23	2.54	0.87
PlatformSegmentation	674	19.42	20.11	21.59	1.02	5.07	19.73	21.99	27.84	3.91	17.78	9.36
ProgressBar	20	7.5	7.8	8.58	0.52	6.67	7.53	7.58	7.62	0.05	0.66	-2.76
RichTextEditor55i	46	7.47	8.49	9.56	1.16	13.66	7.22	8.25	8.89	0.79	9.58	-2.9
Scriptaculous	148	7.44	7.75	8.25	0.36	4.65	7.2	7.4	7.83	0.29	3.92	-4.59
ServiceCenter	5243	67.36	73.42	83.53	7.68	10.46	70.28	71.7	74.86	2.13	2.97	-2.35
SForce	180	12.38	13.77	14.75	1.13	8.21	12.17	13.88	16.39	1.99	14.34	0.74
SimpleInbox	105	10.17	10.52	11.25	0.49	4.66	10.06	10.41	10.7	0.27	2.59	-1.04
SizingCatalog	282	13.95	15.26	16.59	1.36	8.91	13.36	14.5	17.27	1.86	12.83	-4.97
SizingEngine	162	10.17	10.38	10.56	0.2	1.93	10.48	10.65	11.06	0.28	2.63	2.6
SizingValidation	118	9.72	9.92	10.16	0.22	2.22	9.61	9.76	9.94	0.14	1.43	-1.62
Sizing_Schedule	134	10.06	11.05	12.14	1.01	9.14	9.81	11.57	13	1.4	12.10	4.66
Social	24	7.48	7.59	7.75	0.12	1.58	7.55	7.59	7.69	0.07	0.92	0
SolutionSoftwareUnit	18	7.33	7.53	7.84	0.23	3.05	7.31	7.57	8.33	0.51	6.74	0.52
Sorting	98	9	10.49	11.2	1.02	9.72	9.42	10.48	11.67	1.14	10.88	-0.07
TechCenter_Digest	288	14.14	16.07	18.23	2.18	13.57	15.03	16.16	18.06	1.38	8.54	0.61
Training_Management	1722	38.09	39.62	40.8	1.15	2.90	38.95	42.4	52.11	6.48	15.28	7.02
Utility	23	7.41	7.59	7.97	0.26	3.43	7.34	7.46	7.66	0.14	1.88	-1.7
WidgetLibrary	47	8.03	9	9.98	0.9	10.00	7.67	8.56	9.3	0.77	9.00	-4.95
WidgetLibrary40	165	13.31	13.77	14.16	0.38	2.76	13.34	13.88	14.09	0.36	2.59	0.8

Tabela B.1: Comparação entre tempos de compilação

		Compile Time (fixed point algorithm)					Compile Time (cut uses algorithm)				
File (.oml)	**size (Kb)**	**min (s)**	**mean (s)**	**max (s)**	**dev (s)**	**comp %**	**min (s)**	**mean (s)**	**max (s)**	**dev (s)**	**comp %**
aad_Lookups	254	0.02	0.03	0.05	0.01	0.31	0.02	0.02	0.02	0.00	0.21
aad_Resources	387	0.02	0.03	0.05	0.02	0.21	0.02	0.03	0.05	0.01	0.21
aad_Traceability	18	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.14
aad_training	606	0.02	0.02	0.03	0.01	0.13	0.02	0.03	0.05	0.01	0.19
aad_Training_Exams	207	0.02	0.03	0.03	0.01	0.27	0.02	0.02	0.02	0.00	0.18
AuditEvents	104	0.00	0.02	0.03	0.01	0.20	0.00	0.01	0.02	0.01	0.10
CalendarWidget	58	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.13
CaptchaWidget	27	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.12
ChartingServices	2655	0.00	0.02	0.05	0.02	0.20	0.00	0.01	0.02	0.01	0.10
DealReg	368	0.03	0.05	0.08	0.02	0.35	0.02	0.02	0.03	0.01	0.14
DealRegBO	219	0.03	0.04	0.05	0.01	0.28	0.03	0.03	0.03	0.00	0.21
ECHO_UI_v1908	3044	0.95	1.04	1.12	0.07	1.52	0.03	0.04	0.05	0.01	0.06
eMailServices	299	0.00	0.01	0.02	0.01	0.09	0.00	0.01	0.02	0.01	0.09
EM_Impersonate	120	0.00	0.02	0.03	0.01	0.18	0.02	0.02	0.03	0.01	0.18
EnterpriseManager	1129	0.16	0.22	0.27	0.05	0.70	0.02	0.04	0.06	0.02	0.13
FCK_Editor	669	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.11
jQueryTooltip	17	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.13
Licensing	570	0.05	0.08	0.12	0.03	0.36	0.02	0.04	0.05	0.02	0.18
Licensing_SOA	67	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.11
Network	747	0.12	0.16	0.22	0.04	0.88	0.02	0.02	0.03	0.01	0.11
NetworkAccessControl	43	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.11
NetworkDocuments	555	0.09	0.11	0.12	0.01	0.60	0.00	0.02	0.03	0.01	0.11
NetworkMembers	1372	0.52	0.52	0.55	0.02	1.51	0.02	0.04	0.06	0.02	0.12
NetworkSolutions	830	0.08	0.11	0.12	0.02	0.49	0.02	0.04	0.05	0.01	0.18
NetworkSupport	275	0.05	0.08	0.11	0.03	0.54	0.00	0.01	0.02	0.01	0.07
Network_Emails	270	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.08
Paging	26	0.00	0.01	0.02	0.01	0.12	0.00	0.00	0.02	0.01	0.00
Partner	472	0.00	0.02	0.03	0.02	0.10	0.02	0.04	0.05	0.01	0.20
PlatformCommon	119	0.00	0.00	0.02	0.01	0.00	0.02	0.02	0.02	0.00	0.22
PlatformSegmentation	674	0.36	0.39	0.44	0.04	1.77	0.03	0.03	0.03	0.00	0.14
ProgressBar	20	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.13
RichTextEditor55i	46	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.12
Scriptaculous	148	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.01	0.14
ServiceCenter	5243	0.86	0.89	0.95	0.04	1.24	0.02	0.04	0.08	0.03	0.06
SForce	180	0.05	0.07	0.12	0.04	0.50	0.00	0.01	0.02	0.01	0.07
SimpleInbox	105	0.00	0.01	0.02	0.01	0.10	0.00	0.02	0.03	0.01	0.19
SizingCatalog	282	0.02	0.05	0.08	0.03	0.34	0.02	0.02	0.03	0.01	0.14
SizingEngine	162	0.03	0.05	0.06	0.02	0.47	0.00	0.00	0.00	0.00	0.00
SizingValidation	118	0.00	0.03	0.08	0.03	0.31	0.02	0.02	0.03	0.01	0.20
Sizing_Schedule	134	0.00	0.04	0.08	0.03	0.35	0.00	0.01	0.02	0.01	0.09
Social	24	0.00	0.00	0.02	0.01	0.00	0.00	0.00	0.02	0.01	0.00
SolutionSoftwareUnit	18	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.13
Sorting	98	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.10
TechCenter_Digest	288	0.05	0.06	0.08	0.01	0.37	0.00	0.02	0.03	0.01	0.12
Training_Management	1722	0.30	0.37	0.48	0.08	0.87	0.02	0.02	0.05	0.02	0.05
Utility	23	0.00	0.01	0.02	0.01	0.13	0.00	0.00	0.00	0.00	0.00
WidgetLibrary	47	0.00	0.00	0.02	0.01	0.00	0.00	0.01	0.02	0.01	0.12
WidgetLibrary40	165	0.00	0.01	0.02	0.01	0.07	0.00	0.01	0.02	0.01	0.07

Tabela B.2: Comparação dos tempos de execução dos algoritmos

		Memory usage (old version)					Memory usage (new version)					
File (.oml)	**size (Kb)**	**min**	**mean**	**max**	**dev**	**dev %**	**min**	**mean**	**max**	**dev**	**dev %**	**loss %**
aad_Lookups	254	156	156	156	0	0.00	156	156	156	0	0.00	0
aad_Resources	387	173	173	173	0	0.00	173	174.25	178	2.50	1.43	0.72
aad_Traceability	18	152	152	152	0	0.00	152	152	152	0	0.00	0
aad_training	606	186	186	186	0	0.00	186	187.25	191	2.5	1.34	0.67
aad_Training_Exams	207	155	156	157	1.15	0.74	155	156	157	1.15	0.74	0
AuditEvents	104	155	155	155	0	0.00	155	155	155	0	0.00	0
CalendarWidget	58	153	153	153	0	0.00	153	153	153	0	0.00	0
CaptchaWidget	27	154	154	154	0	0.00	154	154	154	0	0.00	0
ChartingServices	2655	170	170.5	172	1	0.59	155	163	170	8.12	4.98	-4.6
DealReg	368	158	158	158	0	0.00	158	158	158	0	0.00	0
DealRegBO	219	158	158	158	0	0.00	158	158	158	0	0.00	0
ECHO_UI_v1908	3044	288	296.25	309	8.96	3.02	294	305.5	314	8.7	2.85	3.03
eMailServices	299	170	170	170	0	0.00	168	169	170	1.15	0.68	-0.59
EM_Impersonate	120	168	169.5	170	1	0.59	168	169	170	1.15	0.68	-0.3
EnterpriseManager	1129	211	223	227	8	3.59	211	215	227	8	3.72	-3.72
FCK_Editor	669	168	168.5	169	0.58	0.34	168	168.5	169	0.58	0.34	0
jQueryTooltip	17	153	153	153	0	0.00	153	153	153	0	0.00	0
Licensing	570	190	190	190	0	0.00	190	190	190	0	0.00	0
Licensing_SOA	67	167	167	167	0	0.00	167	167	167	0	0.00	0
Network	747	158	158	158	0	0.00	158	158	158	0	0.00	0
NetworkAccessControl	43	152	152	152	0	0.00	152	152	152	0	0.00	0
NetworkDocuments	555	162	174	178	8	4.60	162	171.25	183	10.87	6.35	-1.61
NetworkMembers	1372	223	223	223	0	0.00	223	223	223	0	0.00	0
NetworkSolutions	830	190	190	190	0	0.00	190	190	190	0	0.00	0
NetworkSupport	275	173	173	173	0	0.00	173	173.5	175	1	0.58	0.29
Network_Emails	270	157	157	157	0	0.00	157	157	157	0	0.00	0
Paging	26	152	152	152	0	0.00	152	152	152	0	0.00	0
Partner	472	172	172	172	0	0.00	172	177.5	194	11	6.20	3.1
PlatformCommon	119	153	153	153	0	0.00	153	153	153	0	0.00	0
PlatformSegmentation	674	178	178	178	0	0.00	178	178	178	0	0.00	0
ProgressBar	20	152	152	152	0	0.00	152	152	152	0	0.00	0
RichTextEditor55i	46	153	153	153	0	0.00	153	153	153	0	0.00	0
Scriptaculous	148	152	152	152	0	0.00	152	152	152	0	0.00	0
ServiceCenter	5243	326	335	344	9.31	2.78	326	333.75	347	9.18	2.75	-0.37
SForce	180	156	156	156	0	0.00	156	156	156	0	0.00	0
SimpleInbox	105	155	155.25	156	0.5	0.32	155	155.25	156	0.5	0.32	0
SizingCatalog	282	158	158	158	0	0.00	158	158	158	0	0.00	0
SizingEngine	162	157	157	157	0	0.00	157	157.5	159	1	0.63	0.32
SizingValidation	118	154	154	154	0	0.00	154	154	154	0	0.00	0
Sizing_Schedule	134	158	159	160	1.15	0.72	158	159.5	160	1	0.63	0.31
Social	24	153	153	153	0	0.00	153	153	153	0	0.00	0
SolutionSoftwareUnit	18	152	152	152	0	0.00	152	152	152	0	0.00	0
Sorting	98	166	166.75	168	0.96	0.58	166	167	168	1.15	0.69	0.15
TechCenter_Digest	288	158	158	158	0	0.00	158	158	158	0	0.00	0
Training_Management	1722	223	223.25	224	0.5	0.22	223	223.25	224	0.5	0.22	0
Utility	23	152	152	152	0	0.00	152	152	152	0	0.00	0
WidgetLibrary	47	153	153	153	0	0.00	153	153	153	0	0.00	0
WidgetLibrary40	165	189	189	189	0	0.00	189	189.33	190	0.58	0.31	0.17

Tabela B.3: Comparação da utilização de memória entre versões

Definições em *Simple Queries*

O gráfico seguinte dispõe a percentagem de ocorrência de cada um dos tipos de dados envolvidos num total de 2496 *Simple Queries* encontradas em 48 aplicações.

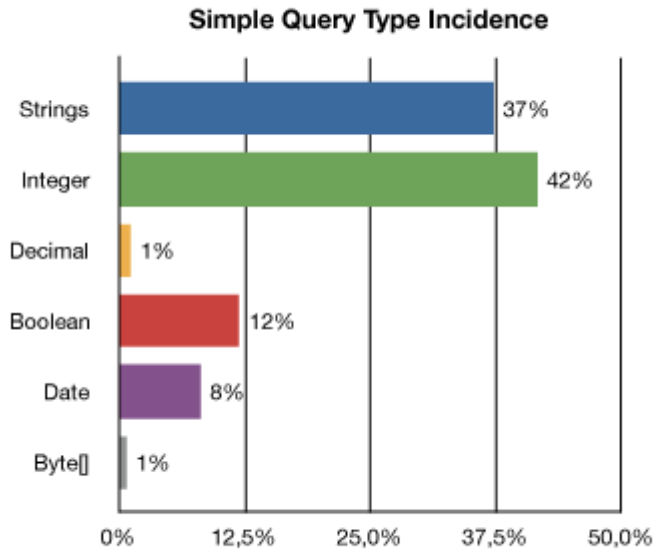

Figura C.1: Percentagem de ocorrência de cada tipo em *Simple Queries*

As próximas tabelas comparam o número de definições de valores em *queries* por cada tipo de dados. Devido ao elevado peso dos tipos *string*, este grupo foi dividido em 4 categorias diferentes (menos do que 50, 200 e 800 caracteres e mais do que 800 caracteres). Em todos os casos é apresentada a percentagem de ganho.

		Query Definitions											
		string < 50			**string < 200**			**string < 800**			**string >= 800**		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	10	8	20.00	33	33	0	0	0	0	12	12	0
aad_Resources	387	6	6	0	41	41	0	21	21	0	18	16	11.11
aad_Traceability	18	0	0	0	0	0	0	0	0	0	0	0	0
aad_training	606	30	29	3.33	51	50	1.96	27	27	0	20	20	0
aad_Training_Exams	207	4	2	50.00	2	2	0	0	0	0	9	7	22.22
AuditEvents	104	6	6	0	9	9	0	1	1	0	3	3	0
CalendarWidget	58	0	0	0	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0	0	0	0	0	0	0
ChartingServices	2655	0	0	0	1	1	0	1	1	0	6	6	0
DealReg	368	0	0	0	32	29	9.38	18	17	5.56	6	5	16.67
DealRegBO	219	0	0	0	20	16	20.00	4	2	50.00	5	2	60.00
ECHO_UI_v1908	3044	244	235	3.69	238	234	1.68	56	53	5.36	3	3	0
eMailServices	299	0	0	0	38	26	31.58	23	15	34.78	18	12	33.33
EM_Impersonate	120	25	12	52.00	16	11	31.25	13	8	38.46	3	1	66.67
EnterpriseManager	1129	101	94	6.93	193	185	4.15	139	130	6.47	5	5	0
FCK_Editor	669	0	0	0	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0	0	0	0
Licensing	570	23	19	17.39	43	43	0	3	3	0	1	1	0
Licensing_SOA	67	0	0	0	0	0	0	0	0	0	0	0	0
Network	747	7	5	28.57	18	12	33.33	20	18	10.00	4	3	25.00
NetworkAccessControl	43	0	0	0	0	0	0	0	0	0	0	0	0
NetworkDocuments	555	18	18	0	15	15	0	36	32	11.11	7	6	14.29
NetworkMembers	1372	107	87	18.69	231	204	11.69	129	118	8.53	19	16	15.79
NetworkSolutions	830	12	11	8.33	54	48	11.11	55	46	16.36	63	29	53.97
NetworkSupport	275	10	10	0	26	26	0	27	27	0	5	5	0
Network_Emails	270	0	0	0	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0	0	0	0
Partner	472	31	6	80.65	60	36	40.00	32	15	53.12	3	2	33.33
PlatformCommon	119	0	0	0	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	109	109	0	54	54	0	108	108	0	29	29	0
ProgressBar	20	0	0	0	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	144	134	6.94	311	290	6.75	92	90	2.17	80	73	8.75
SForce	180	0	0	0	5	5	0	1	1	0	1	1	0
SimpleInbox	105	0	0	0	0	0	0	5	5	0	1	1	0
SizingCatalog	282	33	25	24.24	21	18	14.29	0	0	0	33	28	15.15
SizingEngine	162	0	0	0	0	0	0	0	0	0	0	0	0
SizingValidation	118	1	1	0	0	0	0	1	1	0	2	2	0
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	0	0	0	4	4	0	10	10	0	1	1	0
Training_Management	1722	13	13	0	156	154	1.28	71	71	0	51	50	1.96
Utility	23	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0	0	0	0
Sum		934	830	11.13	1672	1546	7.54	893	820	8.17	408	339	16.91
Average		19.87	17.66	11.13	35.57	32.89	7.54	19.00	17.45	8.17	8.68	7.21	16.91

Tabela C.1: Comparação de definições de *queries* entre versões (1/3)

		Query Definitions											
		total strings			integer			decimal			bool		
File (.oml)	size (Kb)	old	new	gain %	old	new	gain %	old	new	gain %	old	new	gain %
aad_Lookups	254	55	53	3.64	14	14	0	2	2	0	8	8	0
aad_Resources	387	86	84	2.33	63	63	0	0	0	0	30	30	0
aad_Traceability	18	0	0	0	0	0	0	0	0	0	0	0	0
aad_training	606	128	126	1.56	214	209	2.34	4	4	0	47	47	0
aad_Training_Exams	207	15	11	26.67	50	46	8.00	4	4	0	16	14	12.50
AuditEvents	104	19	19	0	15	15	0	2	2	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0	0	0	0	0	0	0
ChartingServices	2655	8	8	0	2	2	0	0	0	0	0	0	0
DealReg	368	56	51	8.93	78	70	10.26	5	4	20.00	2	2	0
DealRegBO	219	29	20	31.03	60	44	26.67	3	0	100.00	4	4	0
ECHO_UI_v1908	3044	541	525	2.96	552	533	3.44	54	51	5.56	182	174	4.40
eMailServices	299	79	53	32.91	43	30	30.23	0	0	0	23	17	26.09
EM_Impersonate	120	57	32	43.86	39	15	61.54	3	0	100.00	8	4	50.00
EnterpriseManager	1129	438	414	5.48	359	354	1.39	4	4	0	131	125	4.58
FCK_Editor	669	0	0	0	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0	0	0	0
Licensing	570	70	66	5.71	126	95	24.60	11	7	36.36	7	4	42.86
Licensing_SOA	67	0	0	0	0	0	0	0	0	0	0	0	0
Network	747	49	38	22.45	43	23	46.51	0	0	0	25	15	40.00
NetworkAccessControl	43	0	0	0	3	3	0	0	0	0	0	0	0
NetworkDocuments	555	76	71	6.58	92	81	11.96	0	0	0	23	21	8.70
NetworkMembers	1372	486	425	12.55	464	402	13.36	8	8	0	122	81	33.61
NetworkSolutions	830	184	134	27.17	145	111	23.45	0	0	0	41	24	41.46
NetworkSupport	275	68	68	0	18	18	0	0	0	0	2	2	0
Network_Emails	270	0	0	0	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0	0	0	0
Partner	472	126	59	53.17	162	125	22.84	0	0	0	36	15	58.33
PlatformCommon	119	0	0	0	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	300	300	0	232	229	1.29	0	0	0	55	55	0
ProgressBar	20	0	0	0	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	627	587	6.38	964	871	9.65	13	13	0	398	312	21.61
SForce	180	7	7	0	0	0	0	0	0	0	2	2	0
SimpleInbox	105	6	6	0	11	11	0	0	0	0	0	0	0
SizingCatalog	282	87	71	18.39	192	164	14.58	4	4	0	20	17	15.00
SizingEngine	162	0	0	0	0	0	0	0	0	0	0	0	0
SizingValidation	118	4	4	0	9	9	0	0	0	0	0	0	0
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	15	15	0	23	23	0	0	0	0	5	5	0
Training_Management	1722	291	288	1.03	387	383	1.03	4	4	0	71	68	4.23
Utility	23	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0	0	0	0
Sum		3907	3535	9.52	4360	3943	9.56	121	107	11.57	1258	1046	16.85
Average		83.13	75.21	9.52	92.77	83.89	9.56	2.57	2.28	11.57	26.77	22.26	16.85

Tabela C.2: Comparação de definições de *queries* entre versões (2/3)

		Query Definitions					
		date			byte[]		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	0	0	0	0	0	0
aad_Resources	387	18	16	11.11	10	10	0
aad_Traceability	18	0	0	0	0	0	0
aad_training	606	53	52	1.89	2	2	0
aad_Training_Exams	207	8	6	25.00	0	0	0
AuditEvents	104	3	3	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0
ChartingServices	2655	0	0	0	0	0	0
DealReg	368	27	24	11.11	0	0	0
DealRegBO	219	18	12	33.33	0	0	0
ECHO_UI_v1908	3044	144	137	4.86	2	2	0
eMailServices	299	11	7	36.36	5	5	0
EM_Impersonate	120	9	4	55.56	0	0	0
EnterpriseManager	1129	51	47	7.84	3	3	0
FCK_Editor	669	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0
Licensing	570	43	19	55.81	0	0	0
Licensing_SOA	67	0	0	0	0	0	0
Network	747	11	5	54.55	0	0	0
NetworkAccessControl	43	0	0	0	0	0	0
NetworkDocuments	555	27	26	3.70	9	9	0
NetworkMembers	1372	92	71	22.83	5	5	0
NetworkSolutions	830	14	12	14.29	7	7	0
NetworkSupport	275	26	26	0	0	0	0
Network_Emails	270	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0
Partner	472	23	10	56.52	0	0	0
PlatformCommon	119	0	0	0	0	0	0
PlatformSegmentation	674	75	75	0	0	0	0
ProgressBar	20	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0
ServiceCenter	5243	88	77	12.50	30	28	6.67
SForce	180	2	2	0	0	0	0
SimpleInbox	105	0	0	0	0	0	0
SizingCatalog	282	14	11	21.43	0	0	0
SizingEngine	162	0	0	0	0	0	0
SizingValidation	118	0	0	0	0	0	0
Sizing_Schedule	134	0	0	0	0	0	0
Social	24	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0
TechCenter_Digest	288	9	9	0	0	0	0
Training_Management	1722	85	83	2.35	3	3	0
Utility	23	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0
	Sum	851	734	13.75	76	74	2.63
	Average	18.11	15.62	13.75	1.62	1.57	2.63

Tabela C.3: Comparação de definições de *queries* entre versões (3/3)

O gráficos seguintes apresentam em forma percentual o ganho obtido em cada tipo.

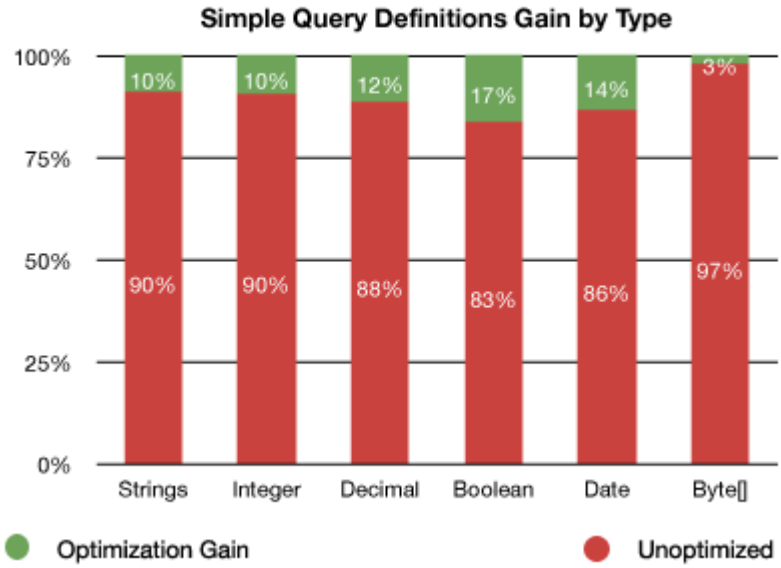

Figura C.2: Ganho em percentagem de definições de *strings* em *queries* nos testes realizados

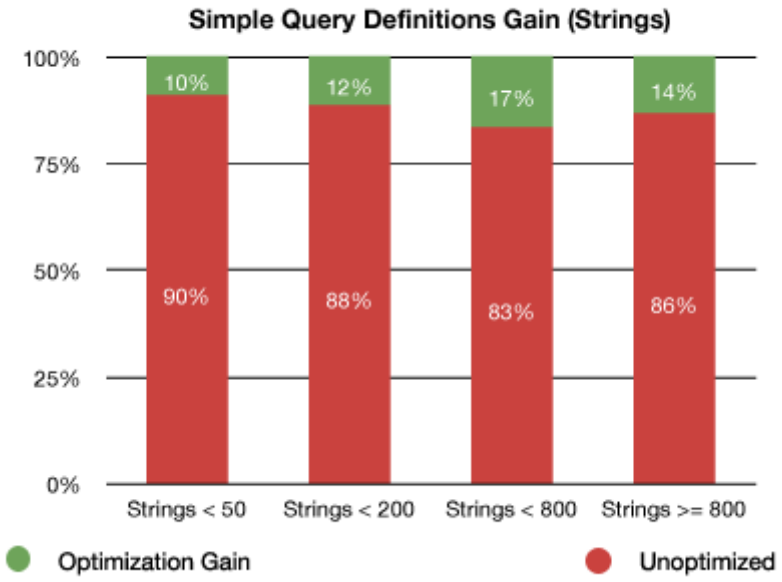

Figura C.3: Ganho em percentagem de definições de *strings* em *queries* nos testes realizados

Usos em *Updates*

O gráfico seguinte dispõe a percentagem de ocorrência de cada um dos tipos de dados envolvidos num total de 248 operações *Update<Entity>* encontradas em 48 aplicações.

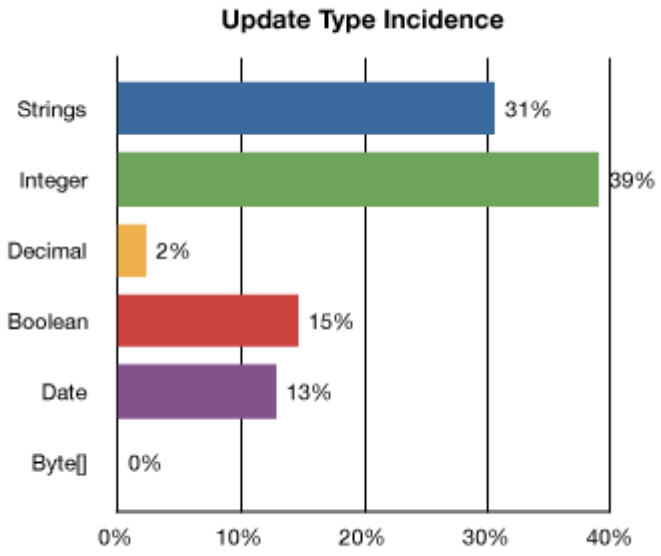

Figura D.1: Percentagem de ocorrência de cada tipo em *Updates*

As próximas tabelas comparam o número de usos em *updates* por cada tipo de dados. Devido ao elevado peso dos tipos *string*, este grupo foi dividido em 4 categorias diferentes (menos do que 50, 200 e 800 caracteres e mais do que 800 caracteres). Em todos os casos é apresentada a percentagem de ganho.

		Update Uses											
		string < 50			string < 200			string < 800			string >= 800		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	2	0	100.00	1	1	0	0	0	0	1	1	0
aad_Resources	387	1	1	0	5	1	80.00	3	1	66.67	3	1	66.67
aad_Traceability	18	0	0	0	0	0	0	0	0	0	0	0	0
aad_training	606	3	1	66.67	6	5	16.67	0	0	0	3	3	0
aad_Training_Exams	207	2	0	100.00	0	0	0	0	0	0	3	0	100.00
AuditEvents	104	0	0	0	0	0	0	0	0	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0	0	0	0	0	0	0
ChartingServices	2655	0	0	0	0	0	0	0	0	0	0	0	0
DealReg	368	0	0	0	6	0	100.00	2	0	100.00	5	2	60.00
DealRegBO	219	0	0	0	9	0	100.00	3	0	100.00	7	2	71.43
ECHO_UI_v1908	3044	29	7	75.86	17	7	58.82	12	6	50.00	1	1	0
eMailServices	299	0	0	0	26	14	46.15	15	0	100.00	16	6	62.50
EM_Impersonate	120	13	0	100.00	9	3	66.67	6	1	83.33	3	1	66.67
EnterpriseManager	1129	32	5	84.38	21	4	80.95	21	0	100.00	1	1	0
FCK_Editor	669	0	0	0	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0	0	0	0
Licensing	570	22	17	22.73	12	12	0	2	2	0	3	3	0
Licensing_SOA	67	0	0	0	0	0	0	0	0	0	0	0	0
Network	747	2	0	100.00	7	1	85.71	2	0	100.00	2	1	50.00
NetworkAccessControl	43	0	0	0	0	0	0	0	0	0	0	0	0
NetworkDocuments	555	0	0	0	2	1	50.00	6	0	100.00	3	1	66.67
NetworkMembers	1372	65	39	40.00	81	34	58.02	47	26	44.68	10	6	40.00
NetworkSolutions	830	6	3	50.00	12	5	58.33	18	6	66.67	55	21	61.82
NetworkSupport	275	0	0	0	0	0	0	0	0	0	0	0	0
Network_Emails	270	0	0	0	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0	0	0	0
Partner	472	27	2	92.59	30	6	80.00	20	3	85.00	3	2	33.33
PlatformCommon	119	0	0	0	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	0	0	0	1	1	0	2	2	0	5	3	40.00
ProgressBar	20	0	0	0	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	32	17	46.88	81	43	46.91	8	6	25.00	30	14	53.33
SForce	180	0	0	0	0	0	0	0	0	0	0	0	0
SimpleInbox	105	0	0	0	0	0	0	10	10	0	1	1	0
SizingCatalog	282	20	7	65.00	12	9	25.00	0	0	0	17	7	58.82
SizingEngine	162	0	0	0	0	0	0	0	0	0	0	0	0
SizingValidation	118	9	9	0	6	6	0	3	3	0	6	6	0
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	0	0	0	0	0	0	1	1	0	1	1	0
Training_Management	1722	0	0	0	48	46	4.17	12	12	0	26	25	3.85
Utility	23	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0	0	0	0
	Sum	265	108	59.25	392	199	49.23	193	79	59.07	205	109	46.83
	Average	5.64	2.30	59.25	8.34	4.23	49.23	4.11	1.68	59.07	4.36	2.32	46.83

Tabela D.1: Comparação no número usos em *Updates* (1/3)

		Update Uses											
		total strings			integer			decimal			bool		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	4	2	50.00	2	2	0	0	0	0	0	0	0
aad_Resources	387	12	4	66.67	8	6	25.00	2	2	0	6	6	0
aad_Traceability	18	0	0	0	0	0	0	0	0	0	0	0	0
aad_training	606	12	9	25.00	27	19	29.63	0	0	0	6	6	0
aad_Training_Exams	207	5	0	100.00	7	3	57.14	0	0	0	3	0	100.00
AuditEvents	104	0	0	0	0	0	0	0	0	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0	0	0	0	0	0	0
ChartingServices	2655	0	0	0	0	0	0	0	0	0	0	0	0
DealReg	368	13	2	84.62	29	13	55.17	3	0	100.00	0	0	0
DealRegBO	219	19	2	89.47	43	19	55.81	4	0	100.00	0	0	0
ECHO_UI_v1908	3044	59	21	64.41	79	41	48.10	9	6	33.33	36	15	58.33
eMailServices	299	57	20	64.91	27	14	48.15	0	0	0	22	13	40.91
EM_Impersonate	120	31	5	83.87	33	9	72.73	3	0	100.00	7	3	57.14
EnterpriseManager	1129	75	10	86.67	40	21	47.50	0	0	0	16	4	75.00
FCK_Editor	669	0	0	0	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0	0	0	0
Licensing	570	39	34	12.82	139	100	28.06	13	9	30.77	24	21	12.50
Licensing_SOA	67	0	0	0	0	0	0	0	0	0	0	0	0
Network	747	13	2	84.62	23	3	86.96	0	0	0	14	3	78.57
NetworkAccessControl	43	0	0	0	0	0	0	0	0	0	0	0	0
NetworkDocuments	555	11	2	81.82	21	9	57.14	0	0	0	7	4	42.86
NetworkMembers	1372	203	105	48.28	184	98	46.74	6	6	0	71	20	71.83
NetworkSolutions	830	91	35	61.54	85	25	70.59	0	0	0	31	11	64.52
NetworkSupport	275	0	0	0	0	0	0	0	0	0	0	0	0
Network_Emails	270	0	0	0	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0	0	0	0
Partner	472	80	13	83.75	61	24	60.66	0	0	0	30	9	70.00
PlatformCommon	119	0	0	0	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	8	6	25.00	13	7	46.15	0	0	0	5	2	60.00
ProgressBar	20	0	0	0	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	151	80	47.02	299	160	46.49	0	0	0	147	55	62.59
SForce	180	0	0	0	0	0	0	0	0	0	0	0	0
SimpleInbox	105	11	11	0	17	17	0	0	0	0	0	0	0
SizingCatalog	282	49	23	53.06	71	32	54.93	2	2	0	14	9	35.71
SizingEngine	162	0	0	0	0	0	0	0	0	0	0	0	0
SizingValidation	118	24	24	0	31	31	0	40	40	0	23	23	0
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	2	2	0	3	3	0	0	0	0	0	0	0
Training_Management	1722	86	83	3.49	103	99	3.88	2	2	0	44	41	6.82
Utility	23	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0	0	0	0
	Sum	1055	495	53.08	1345	755	43.87	84	67	20.24	506	245	51.58
	Average	22.45	10.53	53.08	28.62	16.06	43.87	1.79	1.43	20.24	10.77	5.21	51.58

Tabela D.2: Comparação no número usos em *Updates* (2/3)

		Update Uses					
		date			byte[]		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	0	0	0	0	0	0
aad_Resources	387	3	0	100.00	0	0	0
aad_Traceability	18	0	0	0	0	0	0
aad_training	606	11	8	27.27	0	0	0
aad_Training_Exams	207	3	0	100.00	0	0	0
AuditEvents	104	0	0	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0
ChartingServices	2655	0	0	0	0	0	0
DealReg	368	15	9	40.00	0	0	0
DealRegBO	219	22	13	40.91	0	0	0
ECHO_UI_v1908	3044	33	11	66.67	0	0	0
eMailServices	299	11	4	63.64	0	0	0
EM_Impersonate	120	8	3	62.50	0	0	0
EnterpriseManager	1129	15	4	73.33	0	0	0
FCK_Editor	669	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0
Licensing	570	90	61	32.22	0	0	0
Licensing_SOA	67	0	0	0	0	0	0
Network	747	7	0	100.00	0	0	0
NetworkAccessControl	43	0	0	0	0	0	0
NetworkDocuments	555	4	0	100.00	0	0	0
NetworkMembers	1372	56	29	48.21	0	0	0
NetworkSolutions	830	7	3	57.14	0	0	0
NetworkSupport	275	0	0	0	0	0	0
Network_Emails	270	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0
Partner	472	21	8	61.90	0	0	0
PlatformCommon	119	0	0	0	0	0	0
PlatformSegmentation	674	3	2	33.33	0	0	0
ProgressBar	20	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0
ServiceCenter	5243	56	23	58.93	5	3	40.00
SForce	180	0	0	0	0	0	0
SimpleInbox	105	5	5	0	0	0	0
SizingCatalog	282	11	5	54.55	0	0	0
SizingEngine	162	0	0	0	0	0	0
SizingValidation	118	10	10	0	0	0	0
Sizing_Schedule	134	0	0	0	0	0	0
Social	24	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0
TechCenter_Digest	288	5	5	0	0	0	0
Training_Management	1722	49	47	4.08	0	0	0
Utility	23	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0
	Sum	445	250	43.82	5	3	40.00
	Average	9.47	5.32	43.82	0.11	0.06	40.00

Tabela D.3: Comparação no número usos em *Updates* (3/3)

O gráficos seguintes apresentam em forma percentual o ganho obtido em cada tipo.

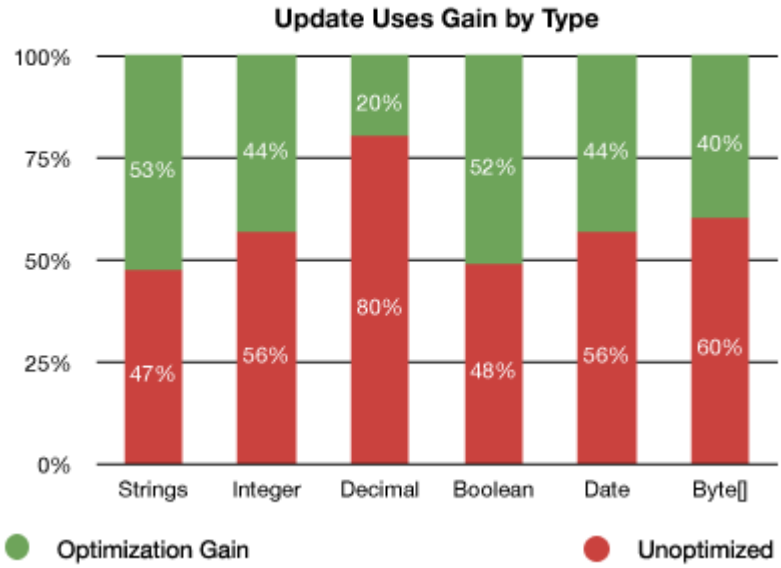

Figura D.2: Ganho em percentagem de usos de *strings* em *Updates* nos testes realizados

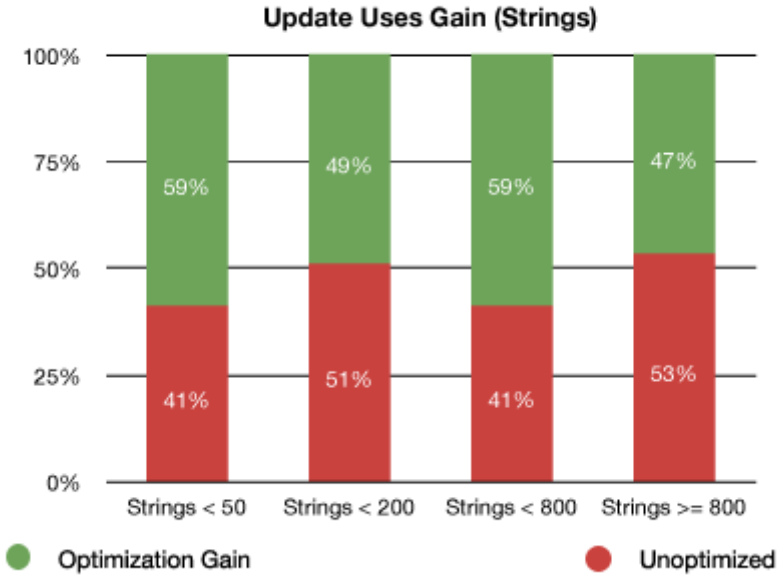

Figura D.3: Ganho em percentagem de usos de *strings* em *Updates* nos testes realizados

Diferenças no *Viewstate*

O gráfico seguinte dispõe a percentagem de ocorrência de cada um dos tipos de dados enviados para o *viewstate* num conjunto de 48 aplicações.

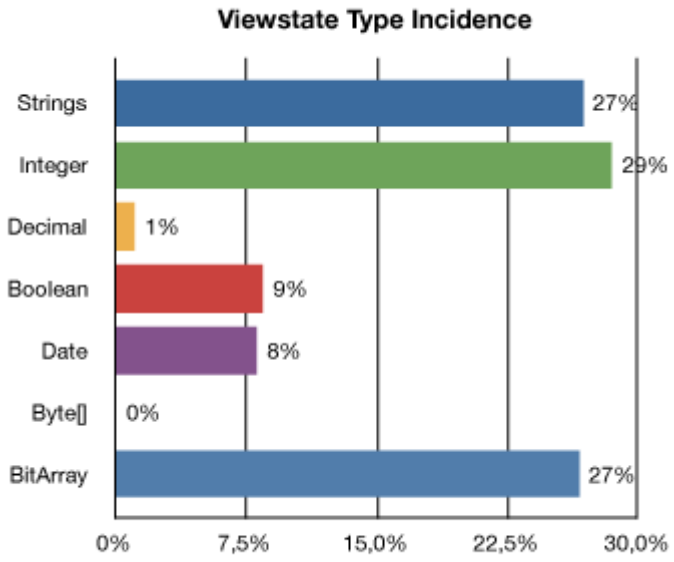

Figura E.1: Percentagem de ocorrência de cada tipo no *Viewstate*

As próximas tabelas comparam o número valores enviados para o *viewstate* por cada tipo de dados. Devido ao elevado peso dos tipos *string*, este grupo foi dividido em 4 categorias diferentes (menos do que 50, 200 e 800 caracteres e mais do que 800 caracteres). Em todos os casos é apresentada a percentagem de ganho.

		Viewstate											
		string < 50			string < 200			string < 800			string >= 800		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	8	4	50.00	16	16	0	0	0	0	5	5	0
aad_Resources	387	6	6	0	18	10	44.44	16	12	25.00	11	7	36.36
aad_Traceability	18	0	0	0	0	0	0	0	0	0	0	0	0
aad_training	606	0	0	0	0	0	0	0	0	0	0	0	0
aad_Training_Exams	207	0	0	0	0	0	0	0	0	0	0	0	0
AuditEvents	104	4	4	0	2	2	0	0	0	0	1	1	0
CalendarWidget	58	0	0	0	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0	0	0	0	0	0	0
ChartingServices	2655	0	0	0	0	0	0	0	0	0	0	0	0
DealReg	368	0	0	0	26	26	0	14	14	0	7	7	0
DealRegBO	219	0	0	0	14	14	0	0	0	0	1	1	0
ECHO_UI_v1908	3044	188	188	0	222	220	0.90	112	110	1.79	15	15	0
eMailServices	299	0	0	0	50	34	32.00	18	2	88.89	21	11	47.62
EM_Impersonate	120	0	0	0	0	0	0	0	0	0	0	0	0
EnterpriseManager	1129	42	42	0	96	96	0	62	62	0	5	5	0
FCK_Editor	669	0	0	0	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0	0	0	0
Licensing	570	8	8	0	22	22	0	0	0	0	1	1	0
Licensing_SOA	67	0	0	0	0	0	0	0	0	0	0	0	0
Network	747	0	0	0	26	26	0	22	22	0	1	1	0
NetworkAccessControl	43	0	0	0	0	0	0	0	0	0	0	0	0
NetworkDocuments	555	18	18	0	20	18	10.00	30	18	40.00	13	9	30.77
NetworkMembers	1372	112	108	3.57	302	294	2.65	134	128	4.48	33	33	0
NetworkSolutions	830	4	2	50.00	14	12	14.29	16	12	25.00	23	19	17.39
NetworkSupport	275	0	0	0	0	0	0	0	0	0	0	0	0
Network_Emails	270	0	0	0	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0	0	0	0
Partner	472	0	0	0	0	0	0	0	0	0	0	0	0
PlatformCommon	119	0	0	0	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	144	144	0	70	70	0	182	182	0	69	69	0
ProgressBar	20	0	0	0	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	164	156	4.88	242	234	3.31	50	48	4.00	55	55	0
SForce	180	0	0	0	0	0	0	0	0	0	0	0	0
SimpleInbox	105	0	0	0	0	0	0	0	0	0	0	0	0
SizingCatalog	282	6	0	100.00	2	0	100.00	0	0	0	5	0	100.00
SizingEngine	162	0	0	0	0	0	0	0	0	0	0	0	0
SizingValidation	118	0	0	0	0	0	0	0	0	0	0	0	0
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	0	0	0	0	0	0	2	2	0	1	1	0
Training_Management	1722	6	6	0	46	46	0	22	22	0	35	35	0
Utility	23	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0	0	0	0
	Sum	710	686	3.38	1188	1140	4.04	680	634	6.76	302	275	8.94
	Average	15.11	14.60	3.38	25.28	24.26	4.04	14.47	13.49	6.76	6.43	5.85	8.94

Tabela E.1: Comparação do tamanho do *Viewstate* das páginas (1/3)

		Viewstate											
		total strings			integer			decimal			bool		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**gain %**
aad_Lookups	254	29	25	13.79	13	13	0	0	0	0	9	9	0
aad_Resources	387	51	35	31.37	21	17	19.05	0	0	0	25	25	0
aad_Traceability	18	0	0	0	0	0	0	0	0	0	0	0	0
aad_training	606	0	0	0	0	0	0	0	0	0	0	0	0
aad_Training_Exams	207	0	0	0	0	0	0	0	0	0	0	0	0
AuditEvents	104	7	7	0	5	5	0	3	3	0	0	0	0
CalendarWidget	58	0	0	0	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	3	3	0	0	0	0	0	0	0
ChartingServices	2655	0	0	0	0	0	0	0	0	0	0	0	0
DealReg	368	47	47	0	37	37	0	5	5	0	0	0	0
DealRegBO	219	15	15	0	11	11	0	3	3	0	0	0	0
ECHO_UI_v1908	3044	537	533	0.74	675	661	2.07	73	73	0	211	201	4.74
eMailServices	299	89	47	47.19	49	33	32.65	0	0	0	29	21	27.59
EM_Impersonate	120	0	0	0	0	0	0	0	0	0	0	0	0
EnterpriseManager	1129	205	205	0	269	269	0	5	5	0	79	79	0
FCK_Editor	669	0	0	0	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0	0	0	0
Licensing	570	31	31	0	63	63	0	13	13	0	7	7	0
Licensing_SOA	67	0	0	0	0	0	0	0	0	0	0	0	0
Network	747	49	49	0	55	55	0	0	0	0	7	7	0
NetworkAccessControl	43	0	0	0	0	0	0	0	0	0	0	0	0
NetworkDocuments	555	81	63	22.22	95	71	25.26	0	0	0	31	27	12.90
NetworkMembers	1372	581	563	3.10	487	473	2.87	15	15	0	157	153	2.55
NetworkSolutions	830	57	45	21.05	81	65	19.75	0	0	0	15	11	26.67
NetworkSupport	275	0	0	0	0	0	0	0	0	0	0	0	0
Network_Emails	270	0	0	0	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0	0	0	0
Partner	472	0	0	0	0	0	0	0	0	0	0	0	0
PlatformCommon	119	0	0	0	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	465	465	0	351	345	1.71	0	0	0	73	69	5.48
ProgressBar	20	0	0	0	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	511	493	3.52	631	577	8.56	0	0	0	223	211	5.38
SForce	180	0	0	0	0	0	0	0	0	0	0	0	0
SimpleInbox	105	0	0	0	0	0	0	0	0	0	0	0	0
SizingCatalog	282	13	0	100.00	21	11	47.62	0	0	0	3	0	100.00
SizingEngine	162	0	0	0	0	0	0	0	0	0	0	0	0
SizingValidation	118	0	0	0	3	3	0	0	0	0	0	0	0
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	3	3	0	7	7	0	0	0	0	0	0	0
Training_Management	1722	109	109	0	189	189	0	3	3	0	49	49	0
Utility	23	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0	0	0	0
	Sum	2880	2735	5.03	3066	2908	5.15	120	120	0	918	869	5.34
	Average	61.28	58.19	5.03	65.23	61.87	5.15	2.55	2.55	0	19.53	18.49	5.34

Tabela E.2: Comparação do tamanho do *Viewstate* das páginas (2/3)

		Viewstate								
		date			byte[]			BitArray		
File (.oml)	**size (Kb)**	**old**	**new**	**gain %**	**old**	**new**	**gain %**	**old**	**new**	**loss %**
aad_Lookups	254	0	0	0	0	0	0	0	25	100.00
aad_Resources	387	17	13	23.53	0	0	0	0	37	100.00
aad_Traceability	18	0	0	0	0	0	0	0	0	0
aad_training	606	0	0	0	0	0	0	0	0	0
aad_Training_Exams	207	0	0	0	0	0	0	0	0	0
AuditEvents	104	0	0	0	0	0	0	0	13	100.00
CalendarWidget	58	0	0	0	0	0	0	0	0	0
CaptchaWidget	27	0	0	0	0	0	0	0	5	100.00
ChartingServices	2655	0	0	0	0	0	0	0	0	0
DealReg	368	27	27	0	0	0	0	0	69	100.00
DealRegBO	219	9	9	0	0	0	0	0	37	100.00
ECHO_UI_v1908	3044	307	299	2.61	0	0	0	0	521	100.00
eMailServices	299	13	5	61.54	0	0	0	0	41	100.00
EM_Impersonate	120	0	0	0	0	0	0	0	0	0
EnterpriseManager	1129	33	33	0	0	0	0	0	205	100.00
FCK_Editor	669	0	0	0	0	0	0	0	0	0
jQueryTooltip	17	0	0	0	0	0	0	0	0	0
Licensing	570	9	9	0	0	0	0	0	49	100.00
Licensing_SOA	67	0	0	0	0	0	0	0	0	0
Network	747	23	23	0	0	0	0	0	49	100.00
NetworkAccessControl	43	0	0	0	0	0	0	0	0	0
NetworkDocuments	555	27	21	22.22	5	5	0	0	113	100.00
NetworkMembers	1372	105	103	1.90	5	5	0	0	325	100.00
NetworkSolutions	830	5	3	40.00	0	0	0	0	77	100.00
NetworkSupport	275	0	0	0	0	0	0	0	0	0
Network_Emails	270	0	0	0	0	0	0	0	0	0
Paging	26	0	0	0	0	0	0	0	0	0
Partner	472	0	0	0	0	0	0	0	0	0
PlatformCommon	119	0	0	0	0	0	0	0	0	0
PlatformSegmentation	674	111	109	1.80	0	0	0	0	305	100.00
ProgressBar	20	0	0	0	0	0	0	0	0	0
RichTextEditor55i	46	0	0	0	0	0	0	0	0	0
Scriptaculous	148	0	0	0	0	0	0	0	0	0
ServiceCenter	5243	137	129	5.84	0	0	0	0	565	100.00
SForce	180	0	0	0	0	0	0	0	0	0
SimpleInbox	105	0	0	0	0	0	0	0	0	0
SizingCatalog	282	3	0	100.00	0	0	0	0	13	100.00
SizingEngine	162	0	0	0	0	0	0	0	0	0
SizingValidation	118	0	0	0	0	0	0	0	5	100.00
Sizing_Schedule	134	0	0	0	0	0	0	0	0	0
Social	24	0	0	0	0	0	0	0	0	0
SolutionSoftwareUnit	18	0	0	0	0	0	0	0	0	0
Sorting	98	0	0	0	0	0	0	0	0	0
TechCenter_Digest	288	9	9	0	0	0	0	0	9	100.00
Training_Management	1722	39	39	0	0	0	0	0	249	100.00
Utility	23	0	0	0	0	0	0	0	0	0
WidgetLibrary	47	0	0	0	0	0	0	0	0	0
WidgetLibrary40	165	0	0	0	0	0	0	0	0	0
	Sum	874	831	4.92	10	10	0	0	2712	100
	Average	18.60	17.68	4.92	0.21	0.21	0	0	57.70	100

Tabela E.3: Comparação do tamanho do *Viewstate* das páginas (3/3)

O gráficos seguintes apresentam em forma percentual o ganho obtido em cada tipo.

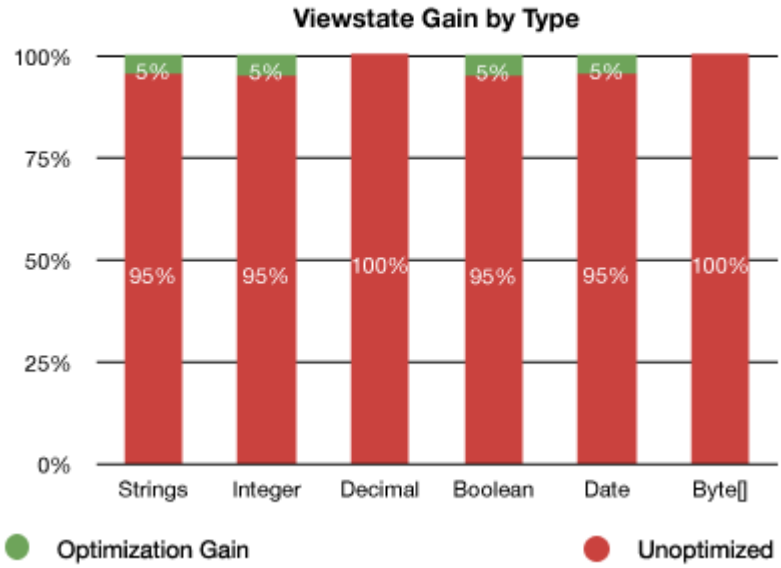

Figura E.2: Ganho em percentagem de *strings* enviadas para o *Viewstate* nos testes realizados

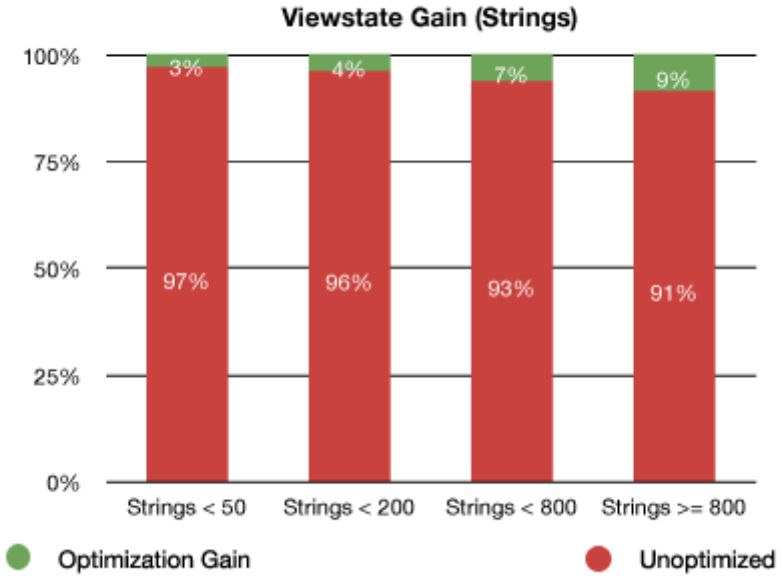

Figura E.3: Ganho em percentagem de *strings* enviadas para o *Viewstate* nos testes realizados

Bibliografia

[1] OutSystems. OutSystems Website, Jul 2009. `http://www.OutSystems.com/`.

[2] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[3] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008.

[4] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, January 1986.

[6] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[8] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[9] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[10] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, December 1997.

[11] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[12] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. Sequel 2: A unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575, 1976.

[13] R. F. Boyce, D. D. Chamberlin, M. M. Hammer, and W. F. King. Specifying queries as relational expressions. *SIGPLAN Not.*, 10(1):31–47, 1975.

[14] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM.

[15] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Science/Engineering/Math, 5 edition, May 2005.

[16] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[17] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[18] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.*, 12(2):197–222, 1994.

[19] P. V. Hall. Optimization of single expressions in a relational data base system. *Journal of Research and Development.*, 20(3):244–257, 1976.

[20] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 311–319, Washington, DC, USA, 1988. IEEE Computer Society.

[21] Matthias Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*, pages 191–205. Springer, 1985.

[22] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[23] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 249–260, New York, NY, USA, 2000. ACM.

[24] Sharma Chakravarthy. Divide and conquer: A basis for augmenting a conventional query optimizer with multiple query proceesing capabilities. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 482–490, Washington, DC, USA, 1991. IEEE Computer Society.

[25] Kian-Lee Tan and Hongjun Lu. Workload scheduling for multiple query processing. *Inf. Process. Lett.*, 55(5):251–257, 1995.

[26] Ben Shneiderman. Batched searching of sequential and tree structured files. *ACM Trans. Database Syst.*, 1(3):268–275, 1976.

[27] K. O'Gorman, A. El Abbadi, and D. Agrawal. Multiple query optimization in middleware using query teamwork. *Softw. Pract. Exper.*, 35(4):361–391, 2005.

[28] Mary Hall, David Padua, and Keshav Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52(2):60–67, 2009.

[29] Microsoft. MSDN SQL Server, Jul 2009. `http://msdn.microsoft.com/en-us/sqlserver/default.aspx`.

[30] Oracle. Oracle Website, Jul 2009. `www.oracle.com/`.

[31] Microsoft. MSDN Bulk Inserts, Jul 2009. `http://msdn.microsoft.com/en-us/library/ms188365.aspx`.

[32] Microsoft. MSDN .NET Development Website, Jul 2009. `http://msdn.microsoft.com/en-us/library/aa139615.aspx`.

[33] Microsoft. MSDN ViewState, Jul 2009. `http://msdn.microsoft.com/en-us/library/ms972976.aspx`.

[34] Sun. Java 2EE, Jul 2009. `java.sun.com/javaee/`.

[35] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Programming Language, The (2nd Edition) (Microsoft .Net Deveolpment Series)*. Addison-Wesley Professional, June 2006.

[36] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[37] Microsoft. MSDN ASP, Jul 2009. `msdn.microsoft.com/en-us/asp.net/default.aspx`.

[38] Mary Wolcott Hall, D. Cooper, Linda Torczon, and Robert Bixby. Managing interprocedural optimization, 1990.

[39] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul A. London. Incremental regression testing, 1993.

[40] Selenium IDE. Selenium IDE Website, Jul 2009. `http://seleniumhq.org/`.

[41] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Alan Apt Series. Prentice Hall, Upper Saddle River, NJ, October 2002.